LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

MIT/LCS/TR-496

AN IN-DEPTH ANALYSIS OF CONCURRENT B-TREE ALGORITHMS

Paul Wang

February 1991

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) MIT/LCS/TR 496	5. MONITORING ORGANIZATION REPORT NUMBER(S) N00014-89-J-1988
6a. NAME OF PERFORMING ORGANIZATION MIT Lab for Computer Science; 6b. OFFICE SYMBOL *(If applicable)*	7a. NAME OF MONITORING ORGANIZATION Office of Naval Research/Dept. of Navy
6c. ADDRESS *(City, State, and ZIP Code)* 545 Technology Square Cambridge, MA 02139	7b. ADDRESS *(City, State, and ZIP Code)* Information Systems Program Arlington, VA 22217
8a. NAME OF FUNDING/SPONSORING ORGANIZATION DARPA/DOD; 8b. OFFICE SYMBOL *(If applicable)*	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS *(City, State, and ZIP Code)* 1400 Wilson Blvd. Arlington, VA 22217	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO / WORK UNIT ACCESSION NO.

11. TITLE *(Include Security Classification)*
An In-Depth Analysis of Concurrent B-Tree Algorithms

12. PERSONAL AUTHOR(S)
Paul Wang

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT *(Year, Month, Day)*	15. PAGE COUNT
Technical	FROM ______ TO ______	February, 1991	131

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
FIELD	GROUP	SUB-GROUP	B-Trees, Parallel algorithms, Dictionaries, Databases, Multi-version memory, Replication, Cache coherency, Software cache management

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

The B-tree is a data structure designed to efficiently support dictionary operations for a variety of applications. In order to increase throughput, many algorithms have been proposed to maintain concurrent operations on B-trees. Replicating objects in memory can play a large role in concurrent B-tree performance, especially for large distributed and parallel systems. Because most replication schemes are *coherent*, readers generally cannot operate concurrently with a writer.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Carol Nicolora	22b. TELEPHONE *(Include Area Code)* (617) 253-5894; 22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
Unclassified

19.

This thesis presents two new concurrent B-tree algorithms. The first is an link algorithm that uses coherent replication; it is based on the Lehman-Yao algorithm which performs better than any other proposed concurrent B-tree algorithm. The second is a similar algorithm that uses multi-version memory, a new semantics for replicated memory. Multi-version memory weakens the semantics of coherent replication by allowing readers to read "old versions" of data. As a result, readers can perform in parallel with a writer. Also, implementations of multi-version memory require less communication and synchronization. Simulation experiments comparing a variety of concurrent B-tree algorithms show that the first algorithm has better performance than previously proposed algorithms and that the second algorithm has significantly better performance and scaling properties than any algorithm using coherent replicated memory.

An In-Depth Analysis of Concurrent B-tree Algorithms

by

Paul Wang

January 1991

This research was supported in part by the National Science Foundation under Grant CCR 8716884, by the Defense Advanced Research Projects Agency (DARPA) under Contract N00014-89-J-1988, and by an equipment grant from Digital Equipment Corporation. The author was supported by a National Science Foundation Graduate Fellowship.

Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge, Massachusetts 02139

An In-Depth Analysis of Concurrent B-tree Algorithms

by

Paul Wang

Abstract

The B-tree is a data structure designed to efficiently support dictionary operations for a variety of applications. In order to increase throughput, many algorithms have been proposed to maintain concurrent operations on B-trees. Replicating objects in memory can play a large role in concurrent B-tree performance, especially for large distributed and parallel systems. Because most replication schemes are *coherent*, readers generally cannot operate concurrently with a writer.

This thesis presents two new concurrent B-tree algorithms. The first is an link algorithm that uses coherent replication; it is based on the Lehman-Yao algorithm which performs better than any other proposed concurrent B-tree algorithm. The second is a similar algorithm that uses multi-version memory, a new semantics for replicated memory. Multi-version memory weakens the semantics of coherent replication by allowing readers to read "old versions" of data. As a result, readers can perform in parallel with a writer. Also, implementations of multi-version memory require less communication and synchronization. Simulation experiments comparing a variety of concurrent B-tree algorithms show that the first algorithm has better performance than previously proposed algorithms and that the second algorithm has significantly better performance and scaling properties than any algorithm using coherent replicated memory.

Keywords: B-Trees, Parallel algorithms, Dictionaries, Databases, Multi-version memory, Replication, Cache coherency, Software cache management

This report is a minor revision of a Master's thesis of the same title submitted to the Department of Electrical Engineering and Computer Science on January 10, 1991, in partial fulfillment of the requirements for the degree of Master of Science in Electrical Engineering and Computer Science. The thesis was supervised by Professor William E. Weihl.

Acknowledgments

First, I thank Professor William Weihl, my thesis supervisor and academic advisor. His insights, comments, and suggestions to this thesis were crucial and greatly appreciated.

I also thank Wilson Hsieh for his ruthless proof-reading and debugging skills. Qin Huang and Brad Spiers waded through portions of my thesis and debugged some of my code. Sanjay Ghemawat repaid an old debt with interest.

I cannot forget to include the members of the programming methodology group. They have made my work environment exciting and unpredictable. I especially thank Kathy Yelick, Earl Waldin, Steve Markowitz, Eric Brewer, and Chris Dellarocas. At some time during the last two years, they were unlucky enough to share an office with me. Also, Carl Waldspurger was always around to initiate serious discussions and not so serious flamefests. Sanjay and Wilson installed addictive computer games on my machine whenever I needed to concentrate. Adrian Colbrook racked up some unbelievable *Battlezone* scores. Bob Gruber's speculations about *Twin Peaks* seemed totally off-base, but turned out to be true. Anthony Joseph ran large batch jobs that competed with my simulations. I never got around to reading Qin's thesis (or Sanjay's, for the matter).

I would like to thank my roommates Kenneth Dandorf and Craig Thompson. Moving next to a video rental store was a mistake. So was getting cable TV. Special appreciation goes to Steve Derezinski for helping me get the biggest hangover in my entire life. I also thank Ronald Carino, Jennifer Harris, Jim Koonmen, Deborah Nungester, Arun Ram, and Tim Shippert, among others. Sweet '54!

Finally, I would like to thank my family, especially my parents. Without their love and support, I would never have been able to finish this thesis.

Contents

List of Figures

Chapter 1

Introduction

The field of computer science, much like the field of mathematics, uses the notion of sets as a fundamental tool. The complexity of many algorithms depends on the efficient implementation of sets. A large class of applications, such as large-scale database systems and symbol tables for compilers, require sets that only need to support the *insert*, *lookup* and *delete* operations. These sets are called *dictionaries*, and the above operations are called *dictionary operations*.

Many data structures have been designed to support dictionary operations. These include *hash tables* (invented, according to Knuth [Knu73], by H. P. Luhn (1953)), *balanced binary trees* [AVL62, Bay72, ST83], and B-trees [BM72]. B-trees are especially useful for applications that use very large dictionaries that are stored in magnetic disks or other direct-access secondary storage devices. The B-tree's structure allows it to minimize the number of disk I/O operations needed to complete an individual dictionary operation.

A recent trend in large-scale computation systems has been a growth in processing power, both within an individual processor, and with the number of processors within a given machine. This growth has led to a greater concern for throughput of data structures. In recent years, many papers (e.g., [BS77, Ell80, KL80, LY81, MR85, Sag86]) have proposed algorithms for maintaining efficient *concurrent operations* on B-trees. Unfortunately, as Johnson and Shasha [JS90] point out, very few studies have thoroughly analyzed the performance of these algorithms. Even less clear are the algorithms' scaling properties. In most analyses, *data contention* has been the main (and sometimes only) concern; such analyses ignore other important issues in large-scale parallel computation.

For example, as parallel and distributed systems become larger and more powerful, their communication networks will become more complicated. Large *network latency* can adversely affect performance. Most B-tree analyses ignore network latency.

Another important and often ignored issue is *resource contention*. For example, every dictionary operation accesses the B-tree's anchor and root. If the system components that store these structures cannot manage the number of requests to access them, then resource contention could become the limiting factor in performance.

Caching and other replication schemes can improve the performance of B-tree algorithms. Caching allows local access to data, thus avoiding network latency. Caching also reduces the dependence of performance on one resource in the system, thus lowering resource contention.

Most caching strategies require expensive communication and synchronization so that reads and writes can appear atomic. We refer to such cache strategies as *coherent shared memory*. In order to improve the scaling properties of coherent shared memory, it becomes necessary to limit the amount of communication and synchronization.

One way to build scalable replicated memory schemes is to loosen the semantics of coherent shared memory. For example, by allowing processes reading data to be assigned an old version of the data, it is possible both to reduce communication and to synchronization needed to implement the replicated memory, and allow readers to access data concurrently with a writer. This weaker semantics exposes the memory replication to the user, and is not as generally useful as the semantics provided by coherent shared memory. However, it turns out that such a scheme is not only adequate for some B-tree algorithms, but it also greatly improves performance and scaling properties.

The goal of this thesis is to analyze various concurrent B-tree algorithms with the above issues in mind. Specifically, the contributions of this thesis are as follows:

- We present a new concurrent B-tree algorithm based on Lehman and Yao's algorithm [LY81] as modified by Sagiv [Sag86]. The Lehman-Yao algorithm is a *link* algorithm, in which each node in the tree contains a pointer to its right neighbor. Because these links allow processes to correct "mistakes" caused by *process overtaking*, the algorithm does not require *lock coupling* to ensure concurrency control. The algorithm also uses a *two-phase* restructuring phase for *insert* operations, so that background processes can perform most of the restructuring needed to balance the tree. Sagiv showed that *insert* and *lookup* operations need only lock one node at a time. Our algorithm extends the *two-phase* approach to the restructuring phase for *delete* operations, so that we can view both *inserts* and *deletes* symmetrically. We base our two-phase *delete* restructuring phase on ideas by Lanin and Shasha [LS86], but with modifications for correctness and efficiency.

- We propose a new semantics for replicated memory, called ***multi-version memory*** [WW90]. This semantics allows a reader to read old versions of replicated data. While less generally useful than ***coherent shared memory***, implementations of multi-version memory produce more concurrency and provide better scaling properties than coherent shared memory. We show how a variety of concurrent dictionary algorithms, including our proposed algorithm above, can use multi-version memory to improve performance. We then present a multi-version memory algorithm based on our algorithm above.

- We compare our two algorithms with algorithms already proposed by others [MR85, BS77]. In our experiments, we measure the performance of various concurrent B-tree algorithms using random operation and key selections, as well as simulations with localized key selections and fixed operation patterns. Using a message-driven simulator for large-scale message-passing architectures, we model resource contention, network latency, and replication, as well as data contention. We find that the performance of the multi-version memory algorithm is significantly better than the other algorithms. Our measurements also indicate that multi-version memory is much more efficient and scalable than coherent shared memory, since it requires less communication and synchronization.

In the next section, we formally define the *dictionary* abstraction. In Section 1.2, we describe the *pseudocode* used in the thesis to describe algorithms. In Section 1.3, we present an overview of the entire thesis.

1.1 Dictionaries

A *dictionary* is a dynamic set (i.e., its elements may change over time) that supports the operations *insert*, *delete*, and *lookup*. Let *Data* denote the set of "data values" that can be stored and maintained by the dictionary. Let *Keys* denote the fully ordered set of values under which the above data values can be "keyed." A dictionary's elements are tuples of the form $< k, d >$, where $k \in Keys$ and $d \in Data$. No two elements of a dictionary have the same key.

We define the specifications for the three dictionary operations as follows:

- *lookup* takes as arguments a dictionary D and a key value k. ***lookup*** returns the data value d, if $< k, d > \in D$. Otherwise, it returns *nil*.

- *insert* takes as arguments a dictionary D, a key value k, and a data value d. *insert* first checks if for all $< k', d' > \in D$, $k \neq k'$. If that is true, then *insert* augments the dictionary D with the element $< k, d >$, otherwise it does nothing.

- *delete* takes as arguments a dictionary D and a key value k. *delete* removes any element $< k, d > \in D$ from D.

We refer to *insert* and *delete* as *update* operations, since they might modify the state of the dictionary.

In specifying the operations, we view each as an atomic action: the implementation must guarantee that the apparent behavior is as if the operations execute atomically in an order consistent with their real-time order. This property is called *linearizability* [HW90].

1.2 Pseudocode

In this thesis, we describe algorithms using a *pseudocode* whose syntax is like C, Algol, or Pascal. The conventions for the pseudocode, based on the conventions for pseudocode used by Cormen, et al. [CLR90], are the following:

- The loop constructs **while** and **for**, and the conditional constructs **if**, **then**, and **else** have the same interpretation as in Pascal.

- **fork** $< op >$ forks a new process, which independently performs $< op >$ in parallel with other processes.

- Assignments are of the form $a := b$, where a is assigned the value of b.

- Array elements are accessed by specifying the array name followed by the index in square brackets. For example, $A[i]$ denotes the i'th element of the array A. We denote the largest and smallest indices of an array A by $A.high$ and $A.low$. Unless specified otherwise, we assume the low bound of array indices is [illegible].

- Compound data will be organized into *records*, which are comprised of *fields*. We access a particular field by specifying the record name, followed by a ".", followed by the field name. For example *rec.foo* refers to the *foo* field of record *rec*. We can concatenate accesses to record fields and array elements; unless otherwise specified, these accesses should be parsed from left to right.

```
proc insertion_sort(A)
% A is an array of integers

    for j = A.low + 1 to A.high do
        key := A[j]
        % insert A[j] into the sorted sequence from A[A.low] to A[j - 1]
        i := j - 1
        while i > A.low - 1 && A[i] > key do
            A[i + 1] := A[i]
            i := i - 1
        end
        A[i + 1] := key
    end
end insertion_sort
```

Figure 1.1: Example pseudocode program.

- A variable representing a record or array is treated as a pointer to data representing the record or array.
- Variables are local to the given procedure. We will not use global variables without explicit indication, and we will denote them by names with all capital letters.
- Sometimes variables will refer to nothing (e.g., unassigned variables, uninitialized pointers). In this case, we give them the special value of *nil*.
- Parameters are passed to a procedure *by value* (i.e. the called procedure receives its own copy of the parameters). When arrays and records are passed, pointers to the data are passed.
- The symbol "%" indicates that the remainder of the line is a comment.

Figure 1.1 is an example program, a simple insertion sort, written in our pseudocode.

1.3 Overview

We organize the thesis as follows.

Chapter 2 presents the B-tree data structure and gives a general overview of the algorithms developed for maintaining concurrent operations on B-trees.

Chapter 3 presents a new concurrent B-tree algorithm based on the Lehman-Yao concurrent B-tree algorithm [LY81] as modified by Sagiv [Sag86]. This algorithm presents a new implementation of the *delete* operation similar to that of Lanin and Shasha's [LS86], but with modifications for correctness and efficiency.

Chapter 4 introduces multi-version memory and shows how the algorithm presented in Chapter 3 can incorporate this novel replication abstraction to produce a more efficient algorithm.

Chapter 5 describes simulation experiments designed to compare the two new algorithms with existing concurrent B-tree algorithms. It discusses how the simulations address issues in analyzing large-scale parallel applications such as data and resource contention, replication, and network latency. It then presents the results of the experiments.

Chapter 6 presents a summary and conclusions. It also describes some directions for future work.

Chapter 2

The Concurrent B-Tree

The *B-tree*, originally proposed by Bayer and McCreight [BM72], is a data structure designed to support dictionary operations. A variant of *2-3 trees* (invented in 1970 by J. E. Hopcroft), the B-tree is well suited for applications where the dynamic set managed by the dictionary is extremely large; such applications must keep the dictionary in secondary storage devices such as magnetic disks. Because accesses to secondary storage are much slower than accesses to real memory, an important factor for performance is the number of I/O operations. Unlike a 2-3 tree, where each non-leaf nodes may only have two or three children, a B-tree node's "maximum fanout" (maximum number of children) can be large. This minimizes the height of the B-tree. In applications that store dictionaries on magnetic disks, it is common for each node in the tree to occupy one page of virtual memory. Therefore, reducing the height of the tree also reduces the number of I/O operations needed to perform dictionary operations. Comer [Com79] presents a full review of B-trees.

Bayer and McCreight [BM72] designed the original B-tree algorithms for sequential applications, where only one process accesses and manipulates the B-tree. The primary concern of such algorithms is minimizing latency. However, in recent years, with the growth of processing power and parallel computing, maximizing throughput has become an important concern.

With the B-tree, it is possible to improve throughput by allowing independent processes to perform concurrent operations. Many proposed algorithms do just that ([BS77, Ell80, KL80, LY81, MR85, Sag86], among others). This chapter presents an overview of these algorithms.

Section 2.1 presents the data structures and abstractions that make up the concurrent B-tree. Section 2.2 describes the existing concurrent B-tree algorithms and discusses the main characteristics that distinguish the algorithms. Section 2.3 discusses other work related to concurrent dictionaries, including data structures other than the B-tree that efficiently support parallel operations.

2.1 B-Tree Data Structures

A B-tree consists of a set of *nodes*. Nodes may either be *leaves*, which store the actual dictionary elements and have no children, or *non-leaves*, which have children and don't store any dictionary elements. Such an arrangement, where only leaves store data, does not correspond to the original design of the B-tree, but a variant commonly referred to as the B+-tree [Com79]. Wedekind [Wed74] pointed out that such a variant is more appropriate for database applications than the original B-tree. This thesis only examine algorithms that maintain B+-trees, and will use the term "B-tree" to mean "B+-tree."

The B-tree *anchor* is a special data structure that contains a pointer to the root of the tree. It might also contain other information, such as the height of the tree, or pointers to other nodes in the tree.

2.1.1 B-Tree Nodes

We define the abstract state of the two types of B-tree nodes as follows:

- A non-leaf node n with j children consists of a sequence $(s_0, p_1, s_1, p_2, s_2, \ldots, p_j, s_j)$, where each p_i is a *downlink*, and each s_i is a *separator*. A downlink is a pointer to a child of n, and a separator is a value in the domain *Keys* used to guide dictionary operations around the tree. The separator values are in ascending order (i.e., $\forall(1 \leq i \leq j), s_{i-1} < s_i$).
- A leaf node n that stores j elements consists of the following:
  - A sequence $(k_1, d_1, k_2, d_2, \ldots, k_j, d_j)$, where each tuple $< k_i, d_i >$ represents an element stored in the leaf. The key values are in ascending order (i.e., $\forall(2 \leq i \leq j), k_{i-1} < k_i$).
  - Two key values k_{min} and k_{max}. For all dictionary elements $< k, d >$ stored in the leaf, $k_{min} < k \leq k_{max}$.

We sometimes compare individual elements in the above sequences "directionally." For example, if an element a occurs before element b in a sequence (e.g., s_0 occurs before

s_1 in the sequence of a non-leaf node), then we sometimes state that "a is to the left of b." Symmetrically, if a occurs after b, then we sometimes state "a is to the right of b."

Some algorithms require the abstract state of nodes to contain more information, such as links to neighbors. We present and define such additions with the presentations of the individual algorithms.

Downlinks connect nodes in a B-tree. If a non-leaf node n stores a downlink in its sequence to node m, we say "n is the parent of m." We cannot arbitrarily assign downlinks. There are a set of restrictions that make the B-tree data structure "legal." Before defining these restrictions, we must first define the following procedures:

- *left_sep* takes as an argument node n. If n is a non-leaf, it returns the leftmost (smallest) separator value stored in n's sequence. If n is a leaf, it returns n's k_{min} value.

- *right_sep* takes a node n. If n is a non-leaf, it returns the rightmost (largest) separator value stored in n's sequence. If n is a leaf, it returns n's k_{max} value.

- *coverset* takes a node n, and returns the set of keys $\{k \mid \textit{left\_sep}(n) < k \leq \textit{right\_sep}(n)\}$.

- *height* takes a node n, and returns the minimum path length from n to a leaf in the tree. (If n is a leaf, $\textit{height}(n) = 0$.)

We now present the restrictions on B-tree nodes that define *legal* states for sequential B-tree algorithms:

- Every node in the tree has exactly one parent (i.e., exactly one other node must have a downlink that points to the node), except the root node, which has no parent.

- If a downlink in n points to the node m, then in the sequence that makes up n's abstract state, the separators to the immediate left and immediate right of the downlink are equal to $\textit{left\_sep}(m)$ and $\textit{right\_sep}(m)$.

- All paths from the root to a leaf node have the same length.

- The coversets of all nodes in any level in the tree form a partition of the keyspace.

- There exists two constants l and u, $l < u$, such that all nodes, except the root, must have at most u and at least l dictionary elements or downlinks (depending on whether the node is a leaf or not). The root must contain at least 2 and at most u downlinks. For most algorithms, l is either 1 or $u/2$.

Most, but not all, concurrent B-tree algorithms follow the first two restrictions. Some, such as the Lehman-Yao algorithm [LY81], have much looser constraints.

We represent a B-tree node n in our pseudocode as a record with the following fields:

- *n.size* stores the number of dictionary elements or downlinks in n.

- *n.level* contains *height*(n).

- If n is a non-leaf with j children, then $n.p$ is an array of downlinks to n's children, and $n.s$ is an array of separators. For $0 \leq a \leq j$, $n.s[a] = s_a$, and for $1 \leq b \leq j$, $n.p[b] = p_b$, where s_a and p_b are separators and downlinks in the sequence of n's abstract state. Note that the minimum index for the array $n.s$ must be 0.

- If n is a leaf storing j dictionary elements, then $n.k$ is an array of key values and $n.d$ is an array of data values. For $1 \leq i \leq j$, $n.k[i] = k_i$ and $n.d[i] = d_i$, where k_i and d_i are keys and data values in the sequence in n's abstract state.

- If n is a leaf, then *n.right_sep* stores *right_sep(n)*. n does not store *left_sep(n)*; its left neighbor stores the value in its *right_sep* field. If n has no left neighbor, we assume *left_sep(n)* is the minimum possible key value.

n may contain other fields as well, depending on the B-tree algorithm. For example, some algorithms require n to have a field *n.rightlink*, which points to n's right neighbor.

Figure 2.1 illustrates the pseudocode representation of B-tree nodes. However, we keep most figures in the thesis simple by drawing a non-leaf node by its abstract state (a sequence of separators and downlinks); we draw a leaf as a sequence of key values, and a right separator value. In order not to clutter figures, we ignore data values.

2.1.2 B-Tree Anchor

The B-tree anchor is a pointer to the root of the tree. We represent the anchor in our pseudocode as a record a with at least two fields:

- *a.root_pointer* is a pointer to the root of the tree.

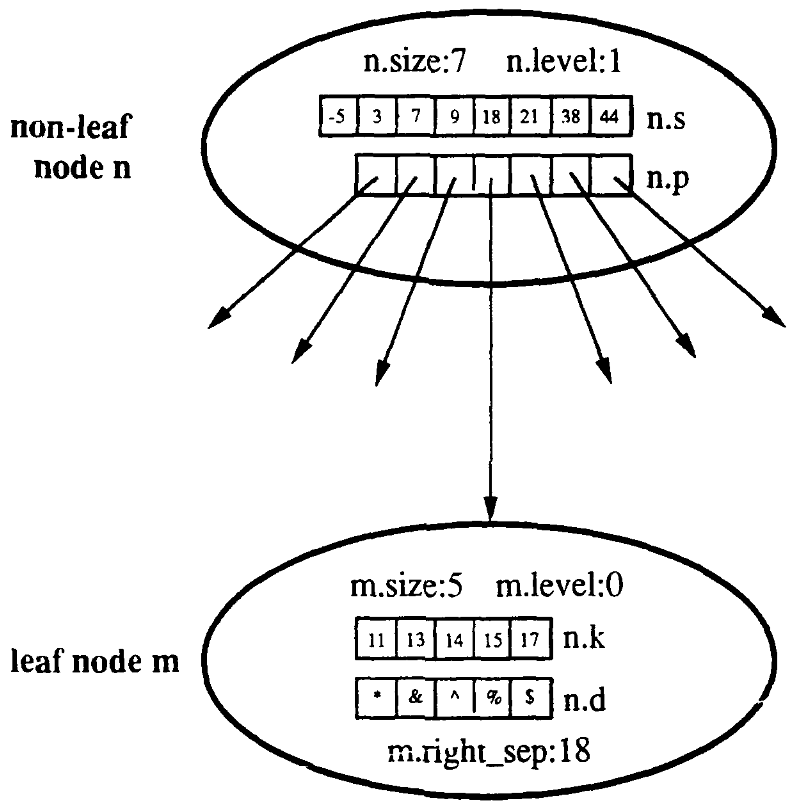


Figure 2.1: B-tree nodes with integer keys.

- *a.root_level* stores the *height* of the node that *a.root_pointer* is pointing to.

For some algorithms, the anchor need not point to the actual root of the tree, but to a node "close" enough to the root to avoid performance degradation. The anchor may store other relevant information as well. For example, some algorithms require the anchor to store an array of pointers that point to the leftmost node of each level in the B-tree.

2.2 Concurrent B-Tree Algorithms

The number of proposed concurrent B-tree algorithms prevents a separate discussion about each algorithm. Instead, this section presents the common issues the algorithms must address, as well the basic distinctions among the algorithms.

All the algorithms share the fundamental problem of *contention*. There are two forms of contention. The first is *data contention*, which forces independent operations to synchronize to prevent them from adversely interfering with each other. The second is *resource contention.* Performance will degrade significantly if too many processes use a single resource in the system (e.g., a memory module in a shared-memory architecture, or a processor in a message-passing machine). Sections 2.2.1 and 2.2.2 discuss the two forms of contention and explain how all the concurrent B-tree algorithms deal with them.

The various algorithms also implement dictionary operations using the same general structure [SG88]. For example, all operations begin with a *descent* from the root of the tree to the proper leaf. They then perform a *decisive operation* (also referred to as a *decisive step*), such as looking up a key in a leaf node, or adding or deleting a dictionary element to or from a leaf. All *update* operations require a *restructuring phase* to ensure that the tree remains balanced. Section 2.2.3 presents these similarities in detail.

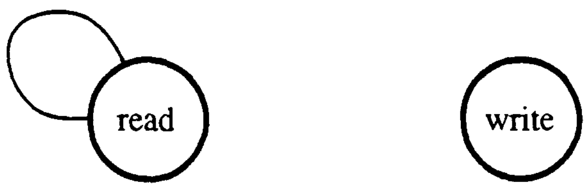
The actual differences among the algorithms lie in the choices made in four orthogonal issues. For some of these issues, such as *conservative vs. optimistic descent*, the optimal choice is clear. Others require more analysis. Section 2.2.4 presents and discusses each of these issues.

2.2.1 Data Contention

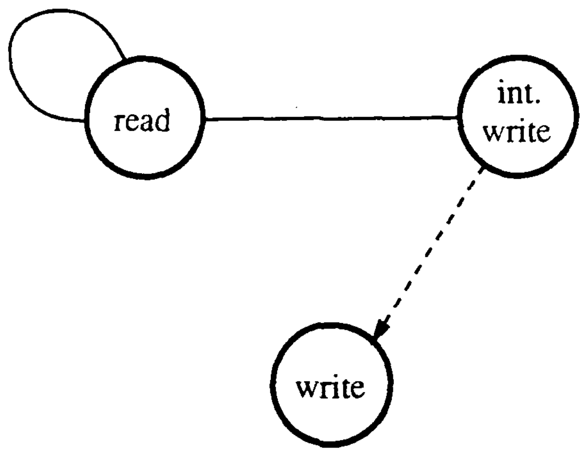
Unless properly synchronized, independent processes accessing a B-tree may adversely interfere with each other. For example, consider two processes, where one is reading data from a B-tree node, and the other is updating the state of the same node. In the middle of its update, the writer may put the abstract (or concrete) state of the node into an improper state, which the reader may read. Preventing this requires synchronization that may cause processes to block one another, thus causing *data contention.*

Concurrency Control

Algorithms must maintain *concurrency control* to prevent adverse interference like the above example. A common approach is to associate a read/write lock with each node in the tree. Independent operations may concurrently acquire the same lock in read mode. However, a process can acquire a lock in write mode only if no other process has acquired the lock in either read or write mode. Figure 2.2(a) shows the *compatibility and convertibility graph* (CCG) [BS77] for read/write locks. A CCG is a directed graph whose nodes are labeled with lock modes and whose edges represent the legal relations between two modes of locks. A solid edge between two nodes denotes the compatibility of two lock modes (i.e., it is possible for two independent processes to concurrently acquire the



(a) read/write locks



(b) read/intention/write locks

Figure 2.2· CCG for various lock protocols.

lock with the modes specified by the nodes). A broken edge from one node to a second indicates that it is legal for a lock of the first type to be directly converted to the second type without releasing the lock. For read/write locks, only readlocks can be acquired concurrently.

We assume a convention where operations to read and write data are distinct from operations for synchronization. The association between the data and the lock that "protects" the data is merely a program convention. To read data, one must first acquire a readlock on the lock associated with the data, read the data, then release the lock. The case of writing to data is analogous. We sometimes refer to acquiring the lock corresponding to a data structure in read (or write) mode as "readlocking (or writelocking the data structure."

Note that we can maintain concurrency control by having only one read/write lock for the entire tree. However, this severely limits the amount of concurrency within the B-tree.

Earlier algorithms [BS77, Ell80, KW82] proposed alternative multi-lock strategies, which included various kinds of "intention to write" locks. Such locks could be held concurrently with readlocks but not with writelocks, or other "intention to write" locks. [1] Figure 2.2(b) shows the CCG of one such multi-lock scheme [BS77]. These strategies turned out to be less effective than more recent algorithms using ordinary read/write locks [LSS87]. Thus, this thesis ignores such lock strategies.

Data Contention and the Root Bottleneck

Maintaining concurrency control causes data contention. Writers block incoming readers and writers from accessing the same B-tree node; readers block incoming writers. Such contention degrades performance, especially when it occurs in the higher nodes in the tree. A process that updates the root or the anchor is especially painful, since every B-tree operation must access both of them. We call this problem the *root bottleneck*.

The approaches used by concurrent B-tree algorithms to reduce data contention, especially the root bottleneck, are the main differences among individual algorithms. Algorithms try to minimize both the time needed to hold writelocks and the number of writelocks a single process may concurrently hold.

2.2.2 Resource Contention

Even if there is no data contention, performance may still degrade as the number of concurrent operations in the B-tree increases. This is due to *resource contention.*

Consider an example where the system stores only one copy of every B-tree node in memory, concurrent processes only read data from the tree, and all processes try to read the same node in the tree. In a shared-memory architecture, all the processes will try to access the same data, which will be located in a single memory module in the machine. In a message-passing architecture, the processor in which the B-tree node resides will receive messages from every process requesting access to the node. In both cases, performance will degrade if the single piece of hardware that maintains the copy of the node cannot handle the number of requests.

Resource contention in a B-tree can be a serious problem, especially for the anchor and the root. Every B-tree operation must visit both. If the system's memory only stores one copy of each node, resource contention will likely be the limiting factor in performance.

[1] Korth [Kor83] introduced similar lock modes, specifically for use in database management.

Coherent Shared Memory

A solution to the resource contention problem is replication. Allowing multiple copies of the same object spreads the work load among many components in the system. Replication can also improve locality. If a copy of an object is kept local to a process that accesses the object, then the process can avoid network delays involved in accessing remote data. One form of replication is hardware caching. In an application such as a concurrent B-tree, caching and other forms of replication are likely to play important roles in improving performance.

Replication schemes that maintain multiple copies of objects generally require *cache coherence* protocols so that individual read and write operations appear atomic.[2] Coherence ensures that the existence of replicated data objects in memory is transparent to the user. We denote the class of memories that use such protocols as *coherent shared memory schemes*. Archibald and Baer [AB86] present an analysis of many proposed coherency algorithms.

Multi-Version Memory

Coherent shared memory allows for better performance by reducing resource contention and improving locality. It also gives the user the appearance that read and write operations are atomic, despite multiple copies of the object. However, the synchronization between readers and writers and the amount of communication between replicated copies grow with both the number of readers and writers, and with the number of copies. By weakening the semantics of coherent shared memory, we can improve the performance of some concurrent B-tree algorithms while still ensuring correctness. We call this new "weakened" memory scheme *multi-version memory*.

A multi-version memory weakens the semantics of a coherent shared memory by allowing a process to read an "old version" of data. (For example, if we use hardware caches, the process might simply use the version in its cache, even if updates by independent processes have not been recorded.) Therefore, individual read and write operations no longer appear atomic. While this semantics is not as generally useful as a coherent shared memory's semantics, many applications can use multi-version memory to improve performance.

[2]Such memory schemes have used many subtly different correctness criteria, including *sequential consistency* [Lam79] and *linearizability* [HW90]. This thesis will use linearizability as its definition of correctness

Multi-version memory achieves better performance than coherent shared memory schemes because of the following important characteristics of its implementations (presented in Chapter 4):

- They allow processes reading data to run concurrently with a process writing to the same data.
- They eliminate "cache misses" resulting from invalidation caused by writes by other processes.
- They eliminate the need for processes to wait for messages that update or invalidate replicated copies.

The first characteristic reduces data contention, since in a multi-version memory, writers do not block readers. The other two characteristics reduce the amount of synchronization and communication needed to maintain the replicated copies, thus reducing the effects of resource contention. Chapter 4 presents multi-version memory in detail, and explains how some concurrent B-tree algorithms can use it to significantly improve performance.

2.2.3 Dictionary Operation Structures

The implementations for the three dictionary operations in all concurrent B-tree algorithms follow a similar structure. This section presents the three basic phases that concurrent B-tree operations use. Note that in the following discussion, we do not take into account concurrency control: we intend for this section to provide a rough framework common to all concurrent B-tree algorithms.

Descent Phase

B-tree operations start with the *descent phase*. Given an operation with key k as an argument, the descent starts at the anchor of the B-tree and continues until the leaf node l, where $k \in \mathit{coverset}(l)$, is reached. The steps of the descent phase are roughly the following:

- Access the anchor to determine the root of the B-tree. The root will be the first node visited.

- At each non-leaf node n visited, find the node m in the level below n such that $k \in coverset(m)$. For most algorithms, this requires finding the appropriate child of n, using the separator values stored in n. This child will be the next node visited.

- When the visited node is l, the descent phase completes.

Decisive Operation

B-tree operations perform a *decisive operation* after the descent phase. *Lookups* check if any dictionary elements stored in l contain key k; *inserts* insert a data element into the leaf; *deletes* delete data elements from the leaf. This thesis will sometimes refer to decisive operations as *decisive steps.*

Restructuring Phase

Update operations (*insert* and *delete*) have one more phase. The *restructuring phase* performs the necessary changes within the B-tree to ensure that the tree stays balanced.

We will describe the restructuring phase of the *insert* operation more closely. (The corresponding phase of the *delete* operation is symmetric, with the concept of "node splitting" replaced with "node merging.") Inserting a dictionary element into the B-tree may cause a leaf to become "full," i.e., the number of elements stored in the leaf exceeds its upper bound. When this happens, the leaf must be *split* in two, with the dictionary elements stored in the original leaf divided up among the two leaves. Figure 2.3 shows how a leaf that is full can be split. We assume that when a leaf is split, the right leaf is a newly created leaf, and the left leaf is the original leaf with its state updated. When such a split occurs, we insert a new downlink and a new separator value into the parent of the original child. This may cause the number of the parent's children to exceed the upper bound, thus forcing the parent to split, and so on. It is possible for this splitting to propagate all the way up to the root of the tree, which causes a new root to be created and the anchor's root pointer to be updated.

Some algorithms require the restructuring phase to be completed before the *update* operation returns. Other algorithms augment the B-tree nodes with additional fields, so that the *update* can return immediately after the decisive operation, and background processes can complete the restructuring phase. Some algorithms "piggyback" the restructuring phase onto the descent phase. The next section will discuss each approach in detail.

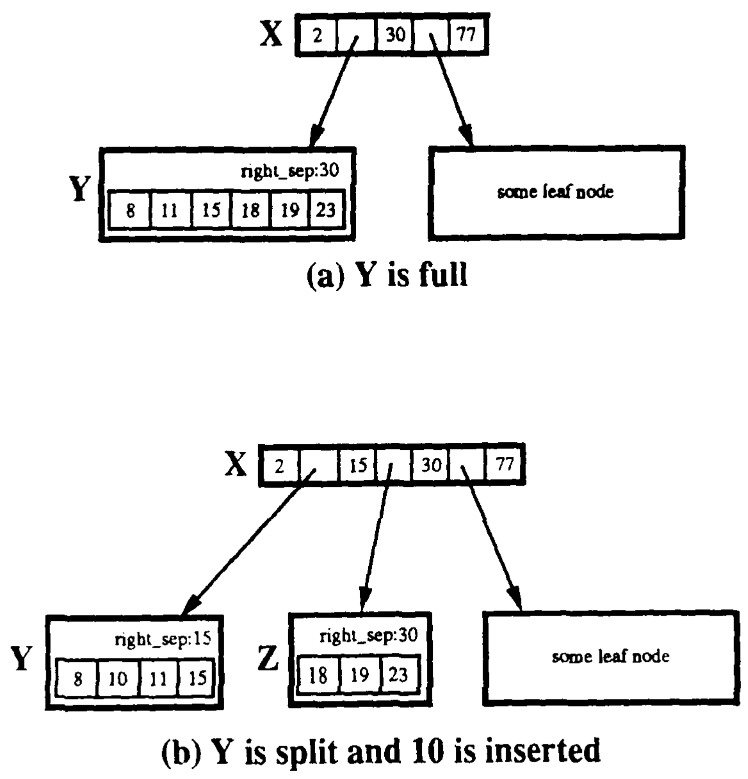


Figure 2.3: Leaf split example.

2.2.4 Issues

The differences between individual concurrent B-tree algorithms lie in the decisions they make in four mostly orthogonal issues. The *lock-coupling vs. link* issue concerns the method that an algorithm uses to control the *process overtaking* problem. *Bottom-up vs. top-down updating* determines the order in which the restructuring phase of *update* operations changes the states of nodes. *Conservative vs. optimistic descent* determines the mode in which the descent phase of *update* operations acquires its locks. Finally, *merge-at-half vs. merge-at-empty* determines when nodes in the B-tree are merged or deleted.

Lock-Coupling vs. Link

Associating read/write locks with B-tree nodes, and accessing the nodes only after acquiring the appropriate lock with the proper mode does not completely solve the concurrency control problem. There are situations, which we refer to as *process overtaking*, where update operations can still adversely affect other concurrent operations.

For example, recall Figure 2.3. Suppose process A, while performing a ***lookup***(19) operation, readlocks node X in Figure 2.3(a) during its descent phase. It concludes that X's child Y is the next node to visit. It releases the readlock on node X. Before A acquires a readlock on Y, process B, which *inserts* key 10, "overtakes" process A, and completes its operation. Since Y will overflow if key 10 is inserted, B's restructuring phase will split Y (into Y and Z), as shown in Figure 2.3(b). When process A eventually readlocks leaf Y, all pertinent information in Y has already been moved to Z, so A accesses the wrong node.

To prevent process overtaking, most B-tree algorithms have their operations use *lock coupling* to block independent operations "above them" from accessing nodes within a sub-tree. During the descent, an operation traverses the tree by first obtaining the appropriate lock on the appropriate child before releasing the lock on the parent. In some cases, descents do not release the lock on the parent until much later. We discuss this in our presentation of the ***bottom-up vs. top-down*** issue.

Lehman and Yao [LY81] suggest another approach. They propose adding *rightlinks* to all nodes. These links are pointers to a node's immediate right neighbor. They effectively eliminate the need for lock-coupling. It is possil for a descent to reach a "wrong node." However, as long as the "wrong node" is to the left of the "correct node," the links provide a way for the operation to redirect itself. In the above example with Figure 2.3(b), if process A readlocks Y and discovers that process B has already moved the relevant contents of Y to Z, it will follow the rightlink from Y to Z.

As pointed out by Sagiv [Sag86], rightlinks allow *insert* and ***lookup*** operations to lock only one node at a time. Lanin and Shasha [LS86] developed similar schemes for *deletes* that lock only one or two nodes concurrently. (Unfortunately, their schemes introduced some errors, which we explain and correct in Chapter 3.)

Rightlinks also allow much of the restructuring phase in update operations to be done by background processes. During an *insert* operation, if a leaf is split, a downlink to the new leaf must be added to the parent. However, with the presence of rightlinks, the insert operation may return after the leaf is split, and a background process may complete the downlink insertion for the parent node.

Consider the following example. In Figure 2.4, we see a sample tree before the insertion of key 10. The insertion will cause the leaf node Y to be split. Figure 2.5 shows the result of the insertion and split. Figure 2.6 shows the insertion into parent X of a downlink pointing to Z (as well as the new separator between Y and Z). However, in a tree with rightlinks, the transformation from Figure 2.4 to Figure 2.6 need not be atomic.

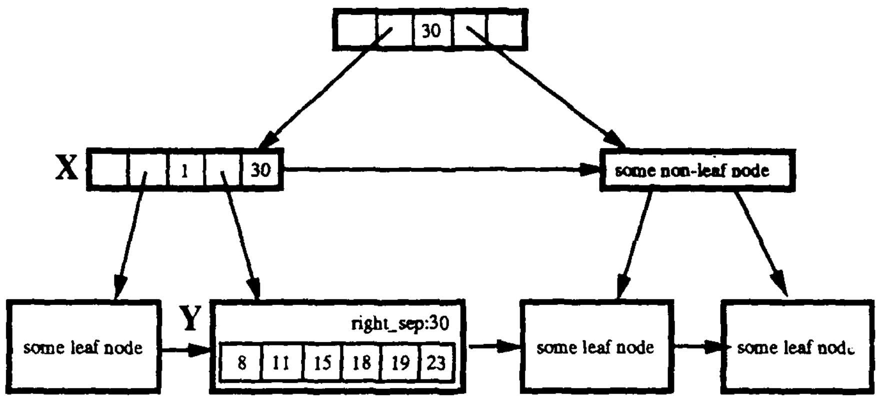


Figure 2.4: Sample Lehman-Yao B-link tree before *inserting* 10.

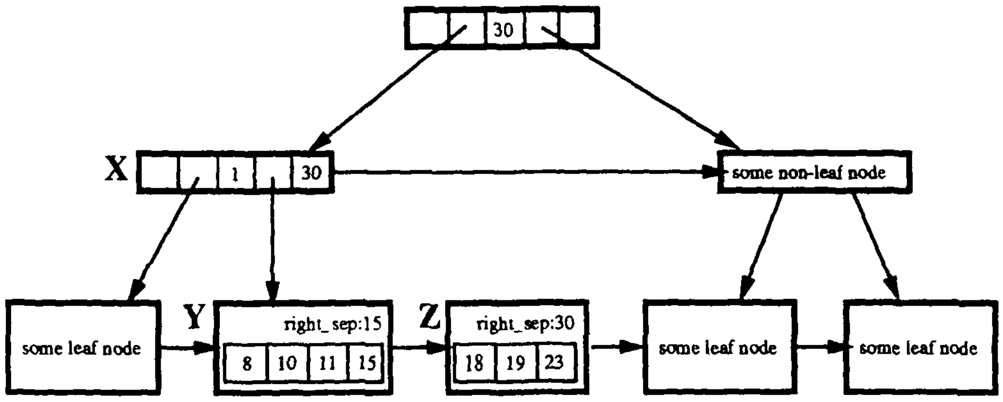


Figure 2.5: Sample Lehman-Yao B-link tree in the middle of *split*.

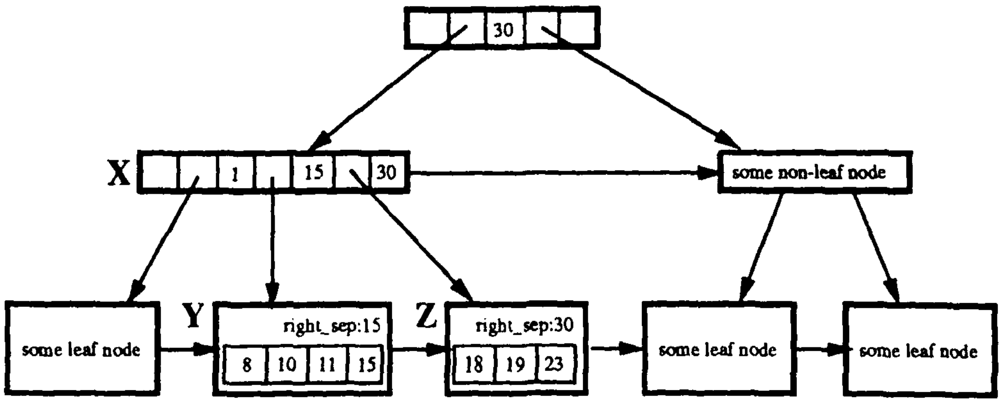


Figure 2.6: Sample Lehman-Yao B-link tree after *inserting* 10.

The tree in Figure 2.5 can adequately support dictionary operations. Any operation that needs to access leaf Z can still do so by visiting Y and chasing its rightlink to Z. We refer to the transformation from Figure 2.4 to Figure 2.5 as a *half_split* and the transformation from Figure 2.5 to Figure 2.6 as a *complete_split*. A background process can do the *complete_split* transformation, so the *insert* operation can complete right after the *half_split*. A delete operation's restructuring phase is similar to that of insertions. In Chapter 3, we discuss in more detail the background transformations for both *update* operations.

Rightlinks eliminate the need for lock-coupling, thus reducing the number of locks that need to be held concurrently. They also allow much of the restructuring phase to be performed in the background, which increases concurrency and throughput. However, traversing rightlinks may also increase latency.

Conservative vs. Optimistic Descent

In the descent phase of *lookup* operations, it is obvious that acquiring readlocks on nodes visited is the correct procedure, since *lookups* do not affect the state of the B-tree. For *update* operations, however, the choice of what type of lock to acquire is not as clear. In a *conservative descent* strategy, an *update* operation writelocks every node it visits during its descent phase, because the restructuring phase may later alter the state of the node.

Bayer and Schkolnick [BS77] originally proposed the idea of *optimistic descent strategies*. These protocols optimistically assume that only leaf nodes need to be restructured during the restructuring phase. Therefore, an *update* operation's descent uses readlocks instead of writelocks, except at the leaf level. If the *update* requires modifications above the leaf level, the optimistic descent gives up, and the *update* retries with a conservative descent.

In general, optimistic descent strategies perform much better than conservative strategies, since they virtually eliminate the need for writelocks in the upper levels of the tree (where contention is highest) [LSS87, JS90]. In most B-tree implementations, the probability of an update operation causing modifications above the leaf level is slight. Lanin et al. [LSS87] predict a probability of $(0.69s)^{-1}$ for B-tree applications with only *inserts* and *lookups*, where s denotes the maximum number of dictionary elements a leaf may hold.

Rightlink algorithms always use an optimistic strategy, and their descents never fail. Since background processes run the restructuring phases and acquire their own locks, there is no need to acquire writelocks during the descent.

Lanin. et al. [LSS87] suggest a simple improvement to the optimistic descent strategy, called the *quick-split.* Parents of leaves, as well as the leaves, require writelocks during the optimistic descent. while the rest still only require readlocks. This change allows the optimistic descent to handle any restructuring in the bottom two levels of the tree, thus further reducing the chances of retrying the *update* with a conservative descent. With a random distribution of operations, the additional writelocks do not significantly affect concurrency within the B-tree; they occur at the low levels in the tree, where contention is usually slight. Simulations of quick-splitting show an improvement in throughput by as much as 20% over that of ordinary optimistic strategies [LSS87].

Bottom-Up vs. Top-Down Updating

To handle restructuring above the leaf level. Bayer and Schkolnick [BS77] describe a *bottom-up* strategy for lock-coupling algorithms, where changes start at the leaf level, and then propagate up the tree. The consequence of such a strategy is that a conservative descent must hold writelocks on all nodes visited until it reaches a descendant that is "safe." We define a safe node as a node where the *update* operation's resulting restructuring phase could not possibly cause it to be split or merged.

Mond and Raz [MR85] propose an alternative *top-down* strategy for lock-coupling protocols that performs the restructuring phase with conservative descents. Before it releases the writelock of a parent, the Mond-Raz strategy writelocks the appropriate child. If required, the Mond-Raz pessimistic descent splits or merges the child and updates the parent's state accordingly. Only after it updates the state of the parent, or if the child did not need updating in the first place, is the parent's writelock released. The Mond-Raz approach "piggybacks" the restructuring phase onto conservative descents. The main advantage to this approach is that it can release a writelock to a node immediately after performing some transformation on one of its children. In contrast, a pessimistic descent with bottom-up restructuring may acquire writelocks for an arbitrary amount of time (until it reach a "safe" descendant).

The *Bottom-up vs. Top-down* issue is relevant only to lock-coupling strategies. It is not an issue for link algorithms. which use optimistic descents and perform restructuring in the background.

For lock-coupling strategies. it is unclear which of the two strategies is more efficient. Bottom-up lock-coupling strategies have the disadvantage of holding writelocks during the descent phase until they reach safe descendants. They also hold more writelocks concurrently. Top-down strategies have longer latencies, since they must check all nodes visited to see if they need restructuring. They may also perform unnecessary work, since all unsafe nodes are restructured regardless of whether or not the *update* actually forces the nodes to be restructured.

Merge-at-Half vs. Merge-at-Empty

Delete operations may reduce the contents of some B-tree nodes to the point where they have to be merged with their siblings in order to maintain balance on the tree. Many B-tree algorithms do not restructure nodes due to underflow conditions until the nodes become empty. We refer to this strategy as *merge-at-empty*. Others use *merge-at-half* protocols that restructure when nodes are half full. Merge-at-half preserves efficient space utilization and keeps the height of the B-tree at $O(\lg n)$, where n is the number of dictionary elements stored in the tree. Merge-at-empty strategies reduce the probability that nodes need to be merged, thus reducing the amount of work performed by restructuring phases of *delete* operations. This lowers data contention with other concurrent processes.

Johnson and Shasha [JS89] discovered that for most concurrent B-tree applications, merge-at-empty produces significantly lower restructuring rates, and only a slightly lower space utilization, than merge-at-half. They concluded that merge-at-empty is a better strategy.

2.3 Related Work

Recent work related to this thesis fall in two basic categories: the analysis of concurrent B-tree algorithms, and the development of new efficient concurrent dictionary algorithms and data structures.

As pointed out by Johnson and Shasha [JS90], there has not been enough work studying the performance of concurrent B-tree algorithms. Bayer and Schkolnick [BS77] and Ellis [Ell80] determine the maximum number of concurrent operations their algorithms can support, but do not predict performance. Analysis by Jipping [JFS85, JFSW90] is very dependent on bus-based architectures, which do not scale well. Lanin, et al. [LSS87] do not allow *delete* operations in their simulations. Lanin and Shasha [LS86] allow *deletes*, but do not take into account resource contention, network latency, or replication.

Johnson and Shasha [JS90] propose a framework for an analytical model to investigate all concurrent B-tree algorithms in a uniform fashion. However, their model also does not take into account network latency or replication. Furthermore, their model assumes the B-tree to be an open system, where the throughput of B-tree operations is equal to the arrival rate of operations at each level in the tree. In applications with high data and resource contention, this assumption may not be valid. For example, Lanin and Shasha [LSS87] explain how high contention may cause a "bursty flow" effect, where large numbers of operations are concentrated at various levels in the tree.

There has been some recent work on developing alternative data structures that support efficient concurrent dictionary operations. Dally [DS85] develops a "rootless" data structure, called the *Balanced Cube*. A collection of nodes connected by a *binary n-cube* communication network, the Balanced Cube avoids bottlenecks with its ability to start dictionary operations at any arbitrary node in the Cube. However, the Cube's performance is very architecture-specific, especially with communication networks. Also, finding efficient methods for dynamically adjusting the Cube's size is a difficult problem.

Herlihy [Her90] proposes a method for transforming sequential data structures to *wait-free* structures using the atomic operation *Compare&Swap*. (Wait-free structures are structures whose operations are guaranteed to complete in a finite number of steps.) He uses this technique to build wait-free concurrent B-trees [Her89]. This work is very recent, and the efficiency and feasibility of such structures in applications is unclear.

Shasha and Goodman [SG88] present a framework for developing and verifying concurrent algorithms for many sequential data structures. Examples include B-trees, hash structures, unordered lists, and other sequential data structures that can support dictionary operations.

Chapter 3

The Coherent Shared Memory Algorithm

In this chapter, we present a new concurrent B-tree algorithm for systems that use coherent shared memory schemes. Because this algorithm uses the *link* method as opposed to the *lock-coupling* method, it locks only one node at a time for *inserts* and *lookups*, and at most two nodes concurrently for *deletes*. Furthermore ne algorithm allows most of the restructuring phase for an *update* operation to be performed after the operation returns. Because of these characteristics, this algorithm's performance should be better than that of any proposed concurrent B-tree algorithm.

Most concurrent B-tree algorithms use lock-coupling to enforce concurrency control. While this technique guarantees correctness, it also sacrifices potential concurrency. Lock-coupling causes entire sub-trees to be excluded from other concurrent processes. Periodically, such algorithms require *update* operations to perform conservative descents, which exclusively lock the root of the tree. This blocks all incoming dictionary operations. Lanin and Shasha [LS86] point out that for this reason, such B-tree algorithms do not have good scaling properties.

In 1981, Lehman and Yao [LY81] introduced an augmented version of the concurrent B-tree, called the *B-link tree*. Such a structure is simply a B-tree with every node augmented by a pointer to its right neighbor. We call these pointers *rightlinks*. Nodes that have no right neighbors have their rightlinks set to *nil*.

The use of rightlinks has two very important results. The first is that it allows concurrent B-link tree algorithms to do away with lock-coupling entirely. As long as descents stray only towards the *left* of the proper path, the rightlinks allows the descents to correct themselves.

The second result is that it allows much of the restructuring phase of *update* operations to be run in the background. Recall the example from Figures 2.4, 2.5, and 2.6. Splitting a node during an *insert* operation can be a *two-phase* procedure. First, a *half_split* transformation splits a node (Figure 2.5). Then, a *complete_split* phase updates the parent of the split node after the *insert* operation returns (Figure 2.6).

The first result increases concurrency by allowing process overtaking, thus reducing the synchronization between independent processes. The second result not only pushes restructuring into the background, but also guarantees that optimistic descents in the Lehman-Yao algorithm are always successful; there is no need to acquire writelocks during descents, since restructuring phases acquire their own locks. This eliminates the need for writelocks on the anchor, the root, or other high-level B-tree nodes during the descent.

Sagiv [Sag86] showed how to implement the Lehman-Yao algorithm such that *lookup* and *insert* operations lock only one node at a time. This further increases concurrency, by minimizing the number of locks that need to be held. Also, Sagiv augmented the B-link tree to improve performance, e.g., Sagiv adds to the anchor a set of pointers to the leftmost node in each tree level.

Unfortunately, Lehman and Yao did not provide for a restructuring phase for *delete* operations; they did not merge under-utilized nodes. Thus their B-link tree data structure would not shrink, even if an application deleted all the dictionary elements in the tree. They proposed that the tree be rebalanced off-line.

Both Salzberg [Sal85] and Sagiv [Sag86] proposed independent background processes, which operate in parallel with processes performing dictionary operations. These processes visit nodes in the tree and perform merge operations on under-utilized nodes. Such solutions, while correct, are not very elegant. It is unclear how the number of processes invoked or the frequency of their invocations affect performance. Also unclear is how different operation patterns can affect the performance of these processes.

Sagiv suggested an alternative approach where the above background processes are created only when leaves become under-utilized. These processes would be removed once the required restructuring is completed. Unfortunately, Sagiv's processes did not merge nodes in a uniform fashion; items were sometimes moved to the right, sometimes to the left. This meant that Sagiv's dictionary operations could "become lost," and descent phases would have to backtrack, or even start over.

Lanin and Shasha [LS86] proposed a restructuring phase for *deletes* that was analogous to Lehman and Yao's two-phase split procedure for *inserts*. First, a *half_merge* transformation merges two nodes. Later, in a background process, a *complete_merge*

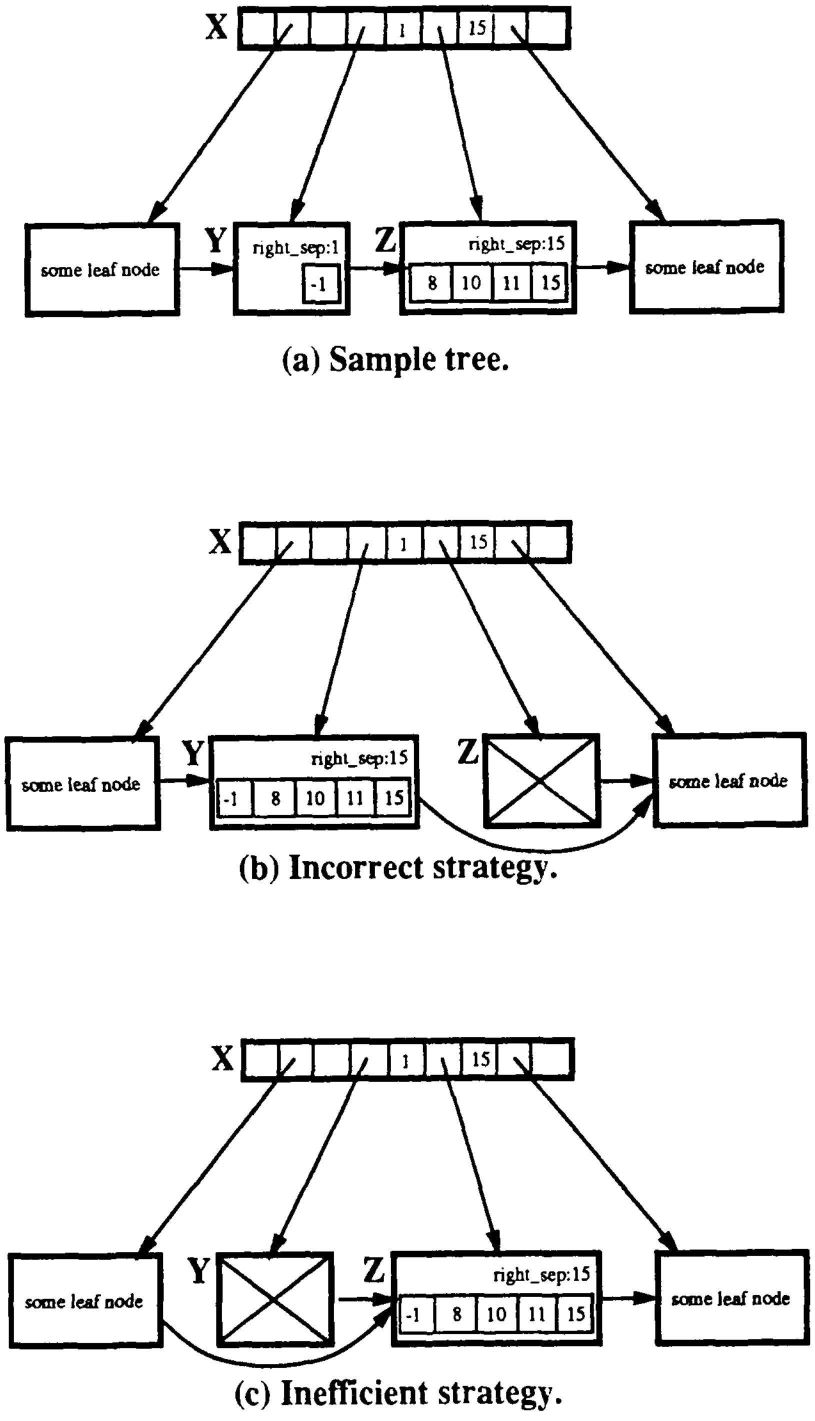


Figure 3.1: Example *half_merge* strategies.

transformation removes from the parent of the merged nodes the downlink pointing to the deleted node.

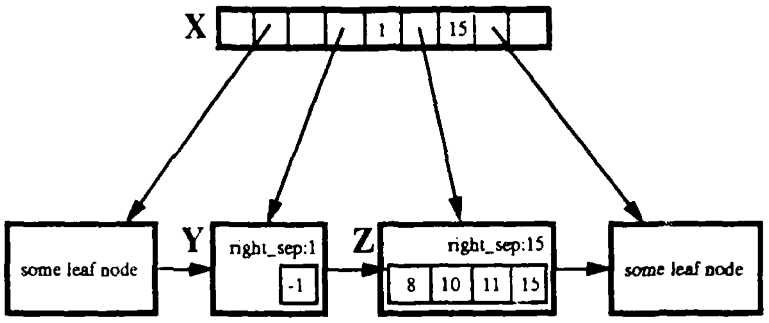
When a node becomes under-utilized, Lanin and Shasha proposed that it be merged with its right neighbor, since rightlinks make the neighbor easy to find. Using the left neighbor would mean either maintaining "leftlinks," or extending the algorithm to search for left neighbors. Given the decision to merge an under-utilized node with its right neighbor, the question remains of exactly how this should be done. Figure 3.1, which shows two possible *half_merge* implementations, illustrates the difficulty in producing correct and efficient *half_merge* and *complete_merge* operations. Nodes that are X'ed out are "marked" as deleted. In Figure 3.1(a), we show a simple tree, where we would like to merge leaves Y and Z. In (b), we move all the contents in Z to Y, and update Y's rightlink to Z's right neighbor. This solution is incorrect, since at the end of the *half_merge* operation, there is no way for processes that access Z to be directed to the proper node Y. In (c), we move all the contents in Y to Z, and update the rightlink of Y's left neighbor to point to Z. This solution is correct, but difficult to implement, since it requires finding and updating Y's left neighbor.

Lanin and Shasha proposed a solution for both the incorrectness problem in (b) and the implementation problem in (c). Consider the example in Figure 3.2. In (a), we present an B-link tree structure. In (b), we show Lanin and Shasha's *half_merge* operation that merges leaves Y and Z. The operation moves data from Z to Y, sets Y's rightlink to Z's right neighbor, and marks Z as deleted. It sets the rightlink of Z to point *left* towards Y, the node to which Z's former contents have been moved. Thus any process that accesses Z after it has been marked as deleted can redirect itself to Y via Z's rightlink. This elegant solution writelocks only two nodes concurrently, and need not search for left neighbors.

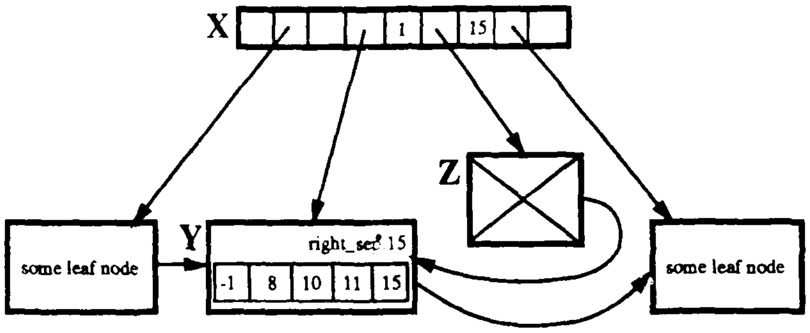
The *complete_merge* operation in this example is straightforward. It locks the parent and removes the downlink that points to the deleted node, as well as the separator to the downlink's immediate left. We see the result of a *complete_merge* in Figure 3.2(c).

Unfortunately the complete algorithm provided by Lanin and Shasha contains a minor error. In addition, other areas in the algorithm can be optimized. In this chapter, we present a complete B-link tree algorithm based on the ideas of Lanin and Shasha, but with the following modifications:

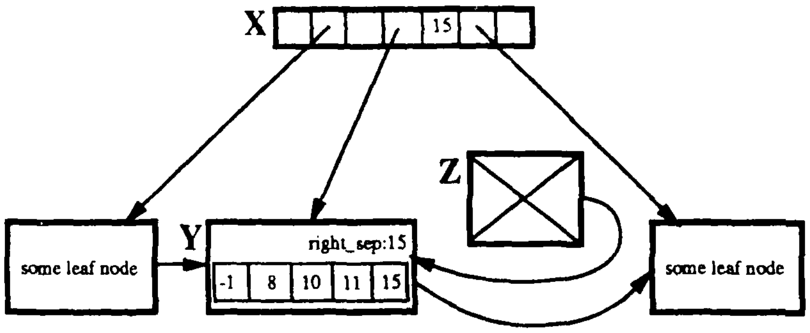
- We present a more efficient approach to maintaining the root pointer in the B-link tree's anchor.



(a) Sample tree.



(b) Half_merge transformation.



(c) Complete_merge transformation

Figure 3.2: Correct merge strategy.

- We propose that left separators of nodes be stored directly in the node. Lanin and Shasha's algorithm requires processes to estimate left separator values based on the states of previously visited nodes. Besides requiring extra overhead, this estimation may sometimes cause restructuring phases of *inserts* and *deletes* to do unnecessary work.

- We discuss an alternative approach to the *complete_merge* operation if the two nodes merged by the *half_merge* have different parents. (The example in Figure 3.2 displayed the more common case where the two merged nodes have the same parent.) Our approach should achieve better performance and use less memory than the one suggested by Lanin and Shasha.

- We explain and correct a problem with Lanin and Shasha's algorithm. The solution requires additional synchronization needed to coordinate independent *complete_split* and *complete_merge* operations.

- We discuss the possibility of maintaining "parent pointers" in each node. We present the advantages of such an approach.

We organize the chapter as follows. Section 3.1 describes the data structures used to implement a B-link tree. The remaining sections present our entire algorithm. Section 3.2 defines a number of procedures used by our algorithm. Sections 3.3, 3.4, and 3.5 present the *lookup*, *insert*, and *delete* operations respectively. Section 3.6 describes the additional synchronization needed to coordinate independent *complete_split* and *complete_merge* operations. Section 3.7 presents the idea and the advantages of maintaining parent pointers at each node. Finally, Section 3.8 summarizes the chapter.

3.1 The B-Link Tree

We construct the data structures for the *B-link tree* by augmenting the data structures for the B-tree. This section presents the extra fields that need to be added to our pseudocode representations of the B-tree data structures.

3.1.1 B-Link Tree Nodes

Each B-link tree node n has the following fields in addition to the ones presented in Section 2.1.1:

```
proc new_node (l)
    % builds new (empty) btree node of level l
    % allocate memory for new node
    node := allocate memory
    % initialize fields
    node.level := l
    node.size := 0
    if l = 0 then
        %% MAX_KEY is global variable denoting largest possible key
        node.right_sep := MAX_KEY
        for i = 1 to MAX_FANOUT do
            node.k[i] := nil
            node.d[i] := nil
        end
    else
        %% MIN_KEY is global variable denoting smallest key
        node.s[0] := MIN_KEY
        for i = 1 to MAX_FANOUT do
            node.s[i] := nil
            node.p[i] := nil
        end
        node.split_waiters := nil
        node.merge_waiters := nil
    end
    node.rightlink := nil
    node.marked? := false
    node.left_most? := false
    return node
end new_node
```

Figure 3.3: *new_node*(l) procedure.

- *n.rightlink* is a pointer to n's right neighbor. If n has no such neighbor, *n.rightlink* is set to *nil*.
- *n.marked?* is a boolean flag that marks deleted nodes. It is initially set to *false*.
- *n.left_most?* is a boolean flag that denotes whether or not n is the leftmost node in its level.
- If n is a non-leaf, then *n.split_waiters* and *n.merge_waiters* are linked lists that are initially set to *nil*. We fully explain these fields in Section 3.6.

The restrictions on the B-link tree nodes in our algorithm are not as stringent as in other B-tree algorithms. Our algorithm does not require that every non-root node in the tree have one parent; some nodes can temporarily have no parents. Also, if a downlink in a non-leaf node points to node n, then the separators stored to the immediate left and right of the downlink are not necessarily equal to *left_sep*(n) and *right_sep*(n), respectively. Our algorithm allows *left_sep*(n) and *right_sep*(n) to be less than or equal to the two separators stored to the downlink's immediate left and right. We discuss these points in more detail during the presentation of the algorithm.

Figure 3.3 presents the pseudocode procedure for creating and initializing new B-link tree nodes. We use the global variable *MAX_FANOUT* to denote the maximum fanout of the tree. We assume the value assigned to *MAX_FANOUT* to be an even integer.

3.1.2 B-Link Tree Anchor

The anchor of a B-link tree is somewhat different from an ordinary B-tree's. A B-link tree anchor a contains the following fields:

- *a.leftmost_nodes* is an array of pointers. The pointer *a.leftmost_nodes*$[i]$ points to the leftmost node in the tree's i'th level. If the tree's height is less than i, then the pointer is set to *nil*.

- *a.root_level* stores the height of the tree's "root."

We do not need the field *a.root_pointer* (described in Section 2.1.2), since the root of the tree is just *a.leftmost_nodes*[*a.root_level*]. Also, *a.root_level* does not necessarily contain the height of the actual root of the tree. As long as *a.root_level* is less than or equal to the actual height of the tree, the algorithm will work properly. The algorithm does makes an effort, for performance's sake, to keep *a.root_level* at, or close to, the actual height of the tree.

Figure 3.4 presents the pseudocode procedures for creating and initializing new B-link trees. In this chapter, we assume that the anchor of a B-link tree is denoted by the global variable *ANCHOR*.

3.2 Miscellaneous Functions

Several functions on B-link tree nodes are used in the rest of the chapter:

- *covers?* takes a node n and a key k, and returns *true* iff $k \in$ *coverset*(n) and *n.marked?* $=$ *false*.

```
proc new_ly_tree ()
     % returns an anchor to an empty B-link Tree.
     % build root
     new_root := new_node(0)
     % build anchor
     anchor := new_anchor()
     anchor.leftmost_nodes[0] := new_root
     return anchor
end new_ly_tree

proc new_anchor ()
     % builds and returns a new btree anchor
     % allocate memory
     anchor := allocate memory
     % initialize fields
     anchor.root_level := 0
     % MAX_HEIGHT is global variable denoting maximum height of tree.
     for i = 0 to MAX_HEIGHT do
          anchor.leftmost_nodes[i] := nil
     end
     return anchor
end new_anchor
```

Figure 3.4: *new_ly_tree*() procedure.

- *successor* takes a non-leaf node n and a key k. If $k > right\_sep(n)$ or $n.marked = true$, then *successor* returns *n.rightlink*. Otherwise, it finds the largest separator s stored in n such that $s < k$, and returns the downlink stored to s's immediate right. If $left\_sep(n) \geq k$, then $successor(n, k)$ is undefined.

- *reaches?* takes a node n and a key k, and returns *true* iff the leaf l that covers k is reachable from n. We formally define "reachable" as follows. Define the function $successor^i$, where $i \geq 0$, as follows:

$$successor^i(n,k) = \begin{cases} n & i = 0, \\ successor(successor^{i-1}(n,k), k) & \text{otherwise.} \end{cases}$$

k is reachable from n, iff for some finite integer j, $successor^j(n,k) = l$. For our algorithm, it turns out that $left\_sep(n) < k$ iff $reaches?(n,k) = true$. (The proof for this has been sketched out, but due to space and time constraints of the thesis, it is not included.) For the rest of the thesis, we will use the above definition of "reachable" (as opposed to "graph reachable" which only checks if nodes are connected via a finite number of edges.)

```
proc ly_lookup (k)
    % readlock leaf that covers k
    node := lookup_descent(k)
            % if k is stored in node, return data.  else return nil
    i := find_key(node, k)
    if i = nil then
        readunlock(node)
        return nil
    end
    answer := i_th_data(node, i)
    readunlock(node)
    return answer
end ly_lookup
```

Figure 3.5: *ly_lookup*(k) procedure.

- *is_leaf?* takes a node and returns *true* iff the node is a leaf.
- *full?* takes a node n, and returns *true* iff the number of dictionary elements or downlinks in n is equal to *MAX_FANOUT*.
- *almost_empty?* takes a node n. If n is a leaf, it returns *true* iff n is not the rightmost leaf and has only one dictionary element stored in it. If n is a non-leaf, it returns *true* iff n only has one downlink.
- *find_key* takes a leaf node l and a key k. It is defined only if $k \in coverset(l)$. If k is the i'th smallest key stored in l, *find_key* returns the index i. Otherwise, it returns *nil*.
- *i_th_data* takes a leaf node l and index i and returns the data associated with the i'th smallest key stored in l.
- *find_child* takes a non-leaf node n, a separator value s, and a downlink to a child node p. If p is the i'th leftmost downlink stored in n and s is the separator stored to p's immediate left, then *find_child* returns i. Otherwise, it returns *nil*.
- *find_sep* takes a non-leaf node n and a separator value s. It returns the integer i iff s is the i'th leftmost separator stored in n, else it returns *nil*.
- *insert_key* takes a leaf node l, a key k, and a data value d. It is defined only if l is not full. *insert_key* inserts the dictionary element $< k, d >$ into l.

```
proc lookup_descent(k)
      % get root of tree
      readlock(ANCHOR)
      level := ANCHOR.root_level
      node := ANCHOR.leftmost_nodes[level]
      readunlock(anchor)
      % descend down tree to leaf level
      readlock(node)
      while ! is_leaf?(node) do
            % find next node to visit
            next := successor(node, k)
            readunlock(node)
            node := next
            readlock(node)
      end
      % move along leaf level to proper leaf, using readlocks
      while ! covers?(node, k) do
            next := node.rightlink
            readunlock(node)
            node := next
            readlock(node)
      end
      return node
end lookup_descent
```

Figure 3.6: *lookup_descent(k)* procedure.

- *insert_child* takes a non-leaf node n, a separator value s, and a downlink to a child node p. It is defined only if n is not full and inserts p into n. *insert_child* inserts s immediately to the left of p.

- *delete_key* takes a leaf node l and an index i, and removes the dictionary element with the i'th smallest key from l.

- *delete_child* takes a non-leaf node n and an index i, and removes the i'th leftmost downlink from n as well as the separator to the immediate left of the downlink.

3.3 The Lookup Operation

In this section, we present the *lookup* operation, shown in Figure 3.5. *ly_lookup(k)* takes as an argument a key k. If the tree contains a dictionary element with key k, then *ly_lookup* returns the element's data value. Otherwise, it returns *nil*. *ly_lookup* first performs the descent phase by calling the procedure *lookup_descent* (line 1 in Figure 3.5).

```
    proc ly_insert (k, d)
        % writelock and return leaf that covers k, using a stack to keep
        % track of path taken
1       stack := new_stack()
2       node := update_descent(k, stack)
        % if k is stored in node, return nil
3       if find_key(node, k) != nil then
4           writeunlock(node)
5           return nil
6       end
        % if node is not full, then insert data into node
7       if ! full?(node) then
8           insert_key(node, k, d)
9           writeunlock(node)
10          return nil
11      end
        % if node is full, then split it
12      return split_leaf(node, k, d, stack)
13  end ly_insert
```

Figure 3.7: *ly_insert*(k, d) procedure.

The procedure *lookup_descent*(k), shown in Figure 3.6, takes as an argument a key k, and readlocks and returns the leaf node that covers k. It first readlocks the anchor and finds the root of the tree (lines 1-4 in Figure 3.6). It then performs two **while** loops to reach the leaf that covers k. The first **while** loop (lines 5-11) uses readlocks and the *successor* function to descend down the tree from the root to the leaf level. The second **while** loop (lines 12-17) uses readlocks and the *covers?* procedure to travel through rightlinks until it finds the leaf that covers k. Note that these loops hold only one readlock at a time. Finally, *lookup_descent* returns the node that covers k (already readlocked) (line 18).

After calling *lookup_descent*, *ly_lookup* performs its decisive operation (lines 2-9 in Figure 3.5). *lookup_descent* has already readlocked the leaf that covers k. If k is not stored in the leaf, then *ly_lookup* unlocks the leaf and returns *nil*. Otherwise, *ly_lookup* unlocks the leaf and returns the data associated with k.

3.4 The Insert Operation

In this section, we present the *insert* operation, shown in Figure 3.7. *ly_insert*(k, d) takes as arguments k and d, where $< k, d >$ is the dictionary element to be inserted. Since *inserts* are more complicated than *lookups*, we divide our discussion among the three

```
    proc update_descent(k, stack)
        % get root of tree
1       readlock(ANCHOR)
2       level := ANCHOR.root_level
3       node := ANCHOR.leftmost_nodes[level]
4       readunlock(ANCHOR)
        % descend to leaf level, using stack to keep track of path
5       readlock(node)
6       while ! is_leaf?(node) do
            % find next node to visit
7           next := successor(node, k)
8           readunlock(node)
9           if next and node are connected via a downlink then
10              push(stack, node)
11          end
12          node := next
13          readlock(node)
14      end
        % move along leaf level to proper leaf, using writelocks
15      readunlock(node)
16      writelock(node)
17      while ! covers?(node, k) do
18          next := node.rightlink
19          writeunlock(node)
20          node := next
21          writelock(node)
22      end
23      return node
24  end update_descent
```

Figure 3.8: *update_descent*($k, stack$) procedure.

phases of the operation.

3.4.1 Descent Phase

ly_insert(k, d) first calls *update_descent* to perform its descent phase (lines 1-2 in Figure 3.7). *update_descent*($k, stack$), shown in Figure 3.8, takes as arguments a key k and a stack *stack*. It writelocks and returns the leaf that covers k, and uses *stack* to record the path taken during the descent phase. *update_descent* first readlocks the anchor and finds the root of the tree (line 1-4 in Figure 3.8). It then uses two **while** loops to reach the leaf that covers k. The first **while** loop (lines 5-14) descends from the root of the tree to the leaf level using readlocks and the *successor* function. Whenever a downlink is traversed, *update_descent* pushes the node last visited in the previous level onto the stack. The second **while** loop (lines 15-22) uses writelocks and the *covers?* function to

```
      proc split_leaf (leaf, k, d, stack)
             % build new split leaf and divide contents of leaf.
 1           new_leaf := divide_leaf(leaf)
             % insert data element into proper leaf
 2           if covers?(leaf, k) then
 3               insert_key(leaf, k, d)
 4           else
 5               insert_key(new_leaf, k, d)
 6           end
             % unlock leaf
 7           new_sep := right_sep(leaf)
 8           writeunlock(leaf)
 9           fork complete_split(new_sep, new_leaf, stack, 1)
10    end split_leaf
```

Figure 3.9: *split_leaf*(*leaf, k, d, stack*) procedure.

traverse rightlinks in the leaf level until it reaches the leaf that covers k.[1]

update_descent returns this leaf (already writelocked) (line 23). Note that this procedure locks only one node at a time.

3.4.2 Decisive Operation

After calling *update_descent*, *ly_insert* performs its decisive operation (lines 3-12 in Figure 3.7). It first checks if the leaf that covers k stores a dictionary element with key k. If such an element already exists, then *ly_insert* unlocks the leaf and returns (lines 3-6). Otherwise, it must insert the element $< k, d >$ into the dictionary. If the number of dictionary elements stored in the leaf is not equal to the upper limit specified by the global variable *MAX_FANOUT*, then *ly_insert* inserts $< k, d >$ into the leaf, unlocks the leaf, and returns (lines 7-11). If the number of elements stored in the leaf is equal to *MAX_FANOUT*, then the leaf must be split to make room for the new dictionary element. *ly_insert* accomplishes this by calling the procedure *split_leaf* (line 12).

split_leaf(*leaf*, k, d, *stack*), shown in Figure 3.9, takes as arguments a leaf *leaf*, key k, data value d, and stack *stack*. It performs a *half_split* on *leaf*, inserts element $< k, d >$ in the appropriate leaf, and then forks an independent process to do a *complete_split* operation.

split_leaf calls *divide_leaf* to split *leaf* in two (line 1 in Figure 3.9). *divide_leaf*(*leaf*),

[1]Alternatively, *ly_insert* could use readlocks until the node that covers s is reached, in which a writelock is then acquired. (After the writelock, *ly_insert* would have to check if the node still covered s.)

```
      proc divide_leaf (leaf)
            % build new leaf
 1          new_leaf := new_node(0)
            % fill new_leaf with right half of leaf
 2          leaf.size := MAX_FANOUT/2
 3          new_leaf.size := MAX_FANOUT/2
            % copy half of array contents to new_leaf
 4          for i = 1 to MAX_FANOUT/2 do
 5              new_leaf.k[i] := leaf.k[i + (MAX_FANOUT/2)]
 6              new_leaf.d[i] := leaf.d[i + (MAX_FANOUT/2)]
 7          end
            % update right_sep values
 8          new_leaf.right_sep := leaf.right_sep
 9          leaf.right_sep := leaf.k[leaf.size]
            % update rightlinks
10          new_leaf.rightlink := leaf.rightlink
11          leaf.rightlink := new_leaf
12          return new_leaf
13    end divide_leaf
```

Figure 3.10: *divide_leaf(leaf)* procedure.

shown in Figure 3.10, takes leaf *leaf* as an argument, and returns a newly created leaf. *divide_leaf* partitions the old contents of *leaf* between *leaf* and the new leaf. It transfers the right half of the dictionary elements in *leaf* to the new leaf (lines 2-7 in Figure 3.10). It then updates the *right_sep* fields of both leaves (lines 8-9). It finally sets *leaf.rightlink* to point to the new leaf, and the new leaf's rightlink to point to the old value of *leaf.rightlink* (line 10-11). *divide_leaf* returns a pointer to the new leaf (line 12).

After calling *divide_leaf*, *split_leaf* inserts the new dictionary element $< k, d >$ into either *leaf* or its new neighbor, depending on which one covers k (lines 2-6 in Figure 3.9). *split_leaf* finally unlocks *leaf* (lines 7-8), and forks a background *complete_split* operation, passing as arguments a pointer to the new leaf and the new separator between *leaf* and the new leaf (line 9).

3.4.3 Restructuring Phase

The restructuring phase for an *insert* operation begins when *split_leaf* forks off an independent process to perform a *complete_split*. *complete_split*$(s, p, stack, l)$, shown in Figure 3.11, takes as arguments a separator value s, downlink p, a stack of node pointers *stack*, and a tree level l. It assumes the node pointers in *stack* point to nodes ordered in consecutive increasing tree level, starting at level l.

```
        proc complete_split (s, p, stack, l)
              % find and writelock the node in the l'th level which covers k
 1            node := find_parent(s, stack, l)
              % if s is already stored in node, that means we have to wait
 2            if find_sep(node, s) != nil then
 3                 push(stack, node)
 4                 insert <s, p, stack> into node.split_waiters
 5                 writeunlock(node)
 6                 return
 7            end
              % check if any waiting operations can be enabled
 8            start_waiters(s, l)
              % if node is not full, insert s and p into node
 9            if ! full?(node) then
10                 insert_child(node, s, p)
                   % check if node could be a new root.  If it is, update the anchor
11                 if new_root?(node) then
12                      fork update_root(node.level)
13                 end
14                 writeunlock(node)
15                 return
16            end
              % else split_interior the node
17            split_interior(node, s, p, stack)
18      end complete_split
```

Figure 3.11: *complete_split*$(s, p, stack, l)$ procedure.

complete_split performs the following three tasks. First, it finds the level l node that covers s. Second, it performs the *complete_split* operation by inserting s and p into the node. If this insertion causes the node to overflow, then the node must be split. To propagate this split, it then invokes a *complete_split* on the next higher level in the tree. Finally, if the above tasks create a new root in the tree, *complete_split* updates the anchor to point to the new root.

Finding the Parent Node

To find the level l node that covers s, *complete_split* calls the procedure *find_parent* (line 1 in Figure 3.11). Figure 3.12 presents pseudocode for *find_parent*$(s, stack, l)$. The procedure takes as arguments a separator value s, a stack of node pointers *stack*, and a tree level l. It writelocks and returns a level l node that covers s.

find_parent first calls the procedure *start_nodes*, which writelocks and returns a level l node n such that *left_sep*$(n) < s$ (line 1 in Figure 3.12). The **while** loop in *find_parent* (lines 2-7) uses writelocks and the function *covers?* to traverse rightlinks to reach the

```
proc find_parent (s, stack, l)
    % get initial node in level l and writelock it
    node := start_node(s, stack, l)
    % move along rightlinks until node covering s is reached
    while ! covers?(node, s) do
        next := node.rightlink
        writeunlock(node)
        node := next
        writelock(node)
    end
    return node
end find_parent
```

Figure 3.12: *find_parent*$(s, stack, l)$ procedure.

node that covers s.[2] This final node is writelocked and returned by *find_parent*.

The procedure *start_node*$(s, stack, l)$, shown in Figure 3.13, takes the same arguments as *find_parent*. It writelocks and returns a level l node n such that *left_sep*$(n) < s$. The node must reach s; otherwise, there will be no way for *find_parent* to find the level l node that covers s. If the stack is not empty, *start_node* pops the topmost node from the stack and writelocks it. If this node reaches s, *start_node* returns the node (lines 1-8 in Figure 3.13). We discuss why the node popped from the stack must be checked to see if it reaches s when we present the *delete* operation.

If the stack is empty (which means either the tree has grown since the descent phase, or a new root needs to be created), or if the node popped from the stack does not reach s, *start_node* will readlock the anchor and find the leftmost node in level l (lines 9-15).[3] If such a node exists, it is writelocked and returned. Otherwise *start_node* creates a new root by calling the procedure *make_root*, and returns this new root. The pseudocode for *make_root* is found in Figure 3.16 and will be discussed later.

Complete_Splitting the Node

After *complete_split*$(s, p, stack, l)$ calls *find_parent* to writelock and return the level l node that covers s (line 1 of Figure 3.11), some steps are taken to coordinate the *complete_split* with other independent *complete_split* and *complete_merge* operations (lines 2-8). For

[2]Alternatively, *find_parent* could use readlocks until the node that covers s is reached, in which a writelock is then acquired. (After the writelock, *find_parent* would have to check if the node still covered s.)

[3]Alternatively, instead of using the leftmost node in l, we can perform a descent from the root using the argument s to find a level l node that reaches s.

```
       proc start_node (s, stack, l)
            % if stack isn't empty, pop the parent
 1          if ! empty_stack?(stack) then
 2               node := pop(stack)
 3               writelock(node)
 4               if reaches?(node, s) then
 5                    return node
 6               end
 7               writeunlock(node)
 8          end
            % lookup ANCHOR. if node is there, return it
 9          readlock(ANCHOR)
10          node := ANCHOR.leftmost_nodes[l]
11          readunlock(ANCHOR)
12          if node != nil then
13               writelock(node)
14               return node
15          end
            % else build new root and return it
16          node := make_root(l)
17          writelock(node)
18          return node
19     end start_node
```

Figure 3.13: *start_node*(*s*, *stack*, *l*) procedure.

now, we will ignore these steps. We discuss them in detail in Section 3.6. If the node returned by *find_parent* is not full, *complete_split* inserts the new separator and downlink (lines 9-10). It then checks if the insertion requires an update to the *root_level* field of the tree's anchor (lines 11-13). We discuss how the anchor field is updated below.

If the node returned by *find_parent* is full, then *complete_split* cannot insert the separator and downlink into the node until it has been split, so *complete_split* calls *split_interior* (line 17 of Figure 3.11). Figures 3.14 and 3.15 present *split_interior* and its accompanying procedure *divide_interior*. These procedures perform *half_split* operations on non-leaf nodes, and are analogous to the procedures *split_leaf* and *divide_leaf*. Note that line 9 in *divide_interior* calls a procedure *divide_waiters* to manipulate the linked lists *node.split_waiters* and *node.merge_waiters*. This procedure will be presented in Section 3.6, which discusses the purpose of these lists. For now, we will ignore them.

Creating a New Root

There are two important issues in the creation of a new root. The first is synchronization: two independent processes should not both create new roots at the same time. The second

```
proc split_interior (node, s, p, stack)
    % build new node
    new_node := divide_interior(node)
    % insert data into proper leaf
    if covers?(node, s) then
        insert_child(node, s, p)
    else
        insert_child(new_node, s, p)
    end
    % unlock node
    new_sep := right_sep(node)
    l := node.level + 1
    writeunlock(node)
    % complete_split
    complete_split(new_sep, new_node, stack, l)
end split_interior
```

Figure 3.14: *split_interior(node, s, p, stack)* procedure.

```
proc divide_interior(node)
    % create new node
    new_node := new_node(node.level)
    % copy half of array contents to new_leaf
    new_node.size := MAX_FANOUT/2
    node.size := MAX_FANOUT/2
    new_node.s[0] = node.s[MAX_FANOUT/2]
    for i = 1 to MAX_FANOUT/2 do
        new_node.s[i] := node.s[i + (MAX_FANOUT/2)]
        new_node.p[i] := node.p[i + (MAX_FANOUT/2)]
    end
    % divide waiters lists
    divide_waiters(node, new_node)
    % set rightlinks
    new_node.rightlink := node.rightlink
    node.rightlink := new_node
    return new_node
end divide_interior
```

Figure 3.15: *divide_interior(node)* procedure.

```
proc make_root (l)
    % writelock anchor
    writelock(ANCHOR)
    % check if node is already there
    if ANCHOR.leftmost_nodes[l] != nil then
        root = ANCHOR.leftmost_nodes[l]
        writeunlock(ANCHOR)
        return(root)
    end
    % build new_root
    new_root := new_node(l)
    new_root.size := 1
    new_root.s[1] := MAX_KEY
    new_root.pointer[1] := ANCHOR.leftmost_nodes[l - 1]
    new_root.left_most? := true
    % place new root in anchor
    ANCHOR.leftmost_nodes[l] := new_root
    writeunlock(ANCHOR)
    return new_root
end make_root
```

Figure 3.16: *make_root(l)* procedure.

is maintaining the *root_level* field in the B-link tree's anchor.

The procedure *make_root(l)*, shown in Figure 3.16, solves the first problem. It takes as an argument an integer *l*, and returns the leftmost B-tree node in level *l*, creating a new root in level *l* if necessary. *make_root* writelocks the anchor and checks for the existence of nodes at level *l* before creating a new root. This protocol prevents independent processes from concurrently creating new roots, since writelocking the anchor sequentializes them.

Maintaining the *root_level* field in the anchor is a separate problem. While the correctness of the our algorithm is ensured as long as there exists a leftmost node in the level specified by the anchor's *root_level* field, the algorithm's efficiency depends on how close *root_level* is to the actual height of the tree. Updating *root_level* during the *make_root* procedure would be correct but inefficient, since the new root only has one downlink, so all descents would chase one extra pointer until a *complete_split* operation on the new root added a second downlink. Instead, we update the anchor's *root_level* during the *complete_split* procedure.

Recall Figure 3.11, which presents the *complete_split* procedure. Right after inserting a separator and a downlink into the node (line 10 in Figure 3.11), *complete_split* checks if the node is a new root that just received a second downlink. It does this by calling the procedure *new_root?* (line 11), which returns *true* iff the node is the only node in its level (i.e., a leftmost node with no right neighbor) and has two downlinks. If *new_root?* returns *true*, then we may have to update the anchor's *root_level* field.

Implementing the update is tricky, in that we must prevent *root_level* from being set to an "outdated value." For example, it is possible for independent processes to add (or delete) further levels to the tree between the time *complete_split* releases the lock on the root and the time the anchor's *root_level* field is updated. Suppose these changes have caused *root_level* to be modified to a more recent value. Then the update corresponding to our *complete_split* procedure might cause *root_level* to point to a level further away from the actual root.

Sagiv [Sag86] suggests that a process maintain a writelock on the root until it updates the anchor's *root_level* field. Therefore, the tree cannot grow or shrink in height until the update has completed. While correct, this solution requires holding writelocks concurrently on both the root and the anchor, the two data structures most commonly accessed in the tree.

Lanin and Shasha propose a separate continuously running background process called the *critic*, which periodically checks the tree for the best *root_level* value. This reduces data contention by allowing the writelock on the root to be released as soon as possible. However, it also forces the maintenance of an independent process, even though it rarely performs useful work. (In most B-tree applications, the probability that an *update* operation will change the height of the tree is slight. For example, Lanin, et al. [LSS87] predict the probability of an *insert* causing the root to split in applications using only *inserts* and *lookups* to be $(0.69 * MAX\_FANOUT)^{-l}$, where l is the height of the tree.)

Rather than maintaining a continuously running process, we suggest invoking such a *critic* whenever needed, and removing it when it has finished its task. In Figure 3.11, during the *complete_split* procedure, if the *new_root?* procedure in line 11 discovers that the anchor's *root_level* field requires updating, *complete_split* will fork off an independent *update_root* process, unlock the node, and return (lines 12-16).[4]

update_root(l), presented in Figure 3.17, writelocks the anchor (line 1 in Figure 3.17) and checks if the new "root level candidate" l is a better root level than the current *ANCHOR.root_level*. If *ANCHOR.leftmost_nodes*[l] = *nil*, then *update_root* does nothing and returns, since the level l does not yet contain any nodes (lines 2-5). If l = *ANCHOR.root_level*, then the procedure also does nothing (lines 6-9). Otherwise it must check if l is indeed a better value for *ANCHOR.root_level*. If $l >$ *ANCHOR.root_level*, then

[4]Alternatively, *complete_split* need not fork off an independent process; it could just run *update_root* directly.

```
       proc update_root (l)
 1         writelock(ANCHOR)
           % check if l has a leftmost node
 2         if ANCHOR.leftmost_nodes[l] = nil then
 3             writeunlock(ANCHOR)
 4             return
 5         end
           % if l = root_level then do nothing
 6         if l = ANCHOR.root_level then
 7             writeunlock(ANCHOR)
 8             return
 9         end
           % readlock node to visit
10         if l > ANCHOR.root_level then
11             candidate := ANCHOR.leftmost_nodes[l]
12         else
13             candidate := ANCHOR.leftmost_nodes[l+1]
14         end
15         readlock(candidate)
           % check if root_level can be updated
16         if (l > ANCHOR.root_level && (! old_root?(candidate))) ||
17            (l < ANCHOR.root_level && old_root?(candidate)) then
18             readunlock(candidate)
19             ANCHOR.root_level := l
20             writeunlock(ANCHOR)
21             return
22         end
23         readunlock(candidate)
24         writeunlock(ANCHOR)
25         return
26     end update_root
```

Figure 3.17: *update_root*(l) procedure.

the update can occur if the leftmost node in level l is not useless. By "useless," we mean the node is the only node in its level and has only one child. If $l < ANCHOR.root\_level$, then the update can occur if the parent of the leftmost node in level l (i.e., the leftmost node in level $l + 1$) is useless. We complete this check by first readlocking the node in question (lines 10-15), then using the procedure *old_root?* (which checks if a node is useless) to determine if the anchor's *root_level* field should be updated (lines 16-26).

We invoke *update_root* whenever a *complete_split* operation detects that the root level might have increased. As we shall see, we also invoke it during *delete* operations, when a *delete* detects that the root level might have decreased. Unlike Lanin and Shasha's *critic*, it is not a continuously running process, but one created on demand. Such a method avoids needless work that a constantly running process might perform while waiting for

```
        proc ly_delete (k)
                % writelock and return leaf that covers k
 1              stack := new_stack()
 2              node := update_descent(k, stack)
                % if k is not stored in node, return nil
 3              index := find_key(node, k)
 4              if index = nil then
 5                  writeunlock(node)
 6                  return nil
 7              end
                % unless node needs to be merged, delete
 8              if ! almost_empty?(node) then
 9                  delete_key(node, index)
10                  writeunlock(node)
11                  return nil
12              end
                % else merge node
13              return merge_leaf(node, stack)
14      end ly_delete
```

Figure 3.18: *ly_delete*(k) procedure.

the tree to grow or shrink.

3.5 The Delete Operation

In this section, we present a *delete* operation that uses a merge-at-empty strategy. As stated in Section 2.2.4, merge-at-empty is suitable for most B-tree applications [JS89]. Figure 3.18 presents the procedure *ly_delete*(k). The procedure takes as an argument k, the key value of the dictionary element we are deleting. The descent phase of the *delete* operation is identical to the descent phase of the *insert* operation; *ly_delete* calls *update_descent*. Therefore in this section, we present only the decisive operation and restructuring phase of *deletes*.

3.5.1 Decisive Operation

ly_delete writelocks the leaf that covers k by calling *update_descent* (lines 1-2 in Figure 3.18). It then performs its decisive operation (lines 3-14). First, it checks if any dictionary element stored in the leaf has the key value k. If there are none, *ly_delete* returns (lines 3-7). If there exists such an element, *ly_delete* uses the procedure *almost_empty?* to check if the leaf is either the rightmost leaf or has more than one data element. If so,

```
  proc merge_leaf (leaf, stack)
        % lock right sibling
1       right_neighbor := leaf.rightlink
2       writelock(right_neighbor)
        % join the leaves, mark right neighbor as deleted
3       old_separator := join_leaves(leaf, right_neighbor)
        % unlock leaves
4       writeunlock(right_neighbor)
5       writeunlock(leaf)
        % Begin Restructuring Phase by forking complete_merge
6       fork complete_merge(old_separator, right_neighbor, stack, 1)
7 end merge_leaf
```

Figure 3.19: *merge_leaf(leaf, stack)* procedure.

the data element is simply deleted and *ly_delete* returns (lines 8-12). Otherwise, the leaf has to be merged with its right neighbor, since removing its only data element would leave it empty. To perform the merge. *ly_delete* calls the procedure *merge_leaf.*

Note that we have decided not to merge the rightmost leaf, even if it becomes empty. (Recall that *almost_empty?* returns *false* if the node has no right neighbor.) Trying to merge it with its left neighbor would be inconvenient and would require extra locks to be held. Not deleting it may cause under-utilization of the B-link tree data structure. However, such under-utilization will probably be slight, especially for applications where the B-tree is growing over time.

Half_Merging Leaves

merge_leaf(leaf, stack), shown in Figure 3.19, takes as arguments a leaf node *leaf* and a stack *stack.* It assumes *leaf* has already been writelocked, and performs a *half_merge* operation on *leaf* and its right neighbor after deleting from *leaf* all of its contents. It first writelocks the right neighbor (lines 1-2 of Figure 3.19). It then calls the procedure *join_leaves* (line 3).

join_leaves(left_leaf, right_leaf), shown in Figure 3.20, takes as arguments two neighbor leaves. It first saves the old separator value between the two leaves (line 1 in Figure 3.20). It then moves the right separator and dictionary elements of the right leaf into the left leaf (lines 2-7). Note that *join_leaves* "overwrites" the left leaf's former contents. It then sets the rightlink of the left leaf to the rightlink of the right leaf and the rightlink of the right leaf to the left leaf (lines 8-9). Finally, *join_leaves* marks the right leaf as deleted, and returns the old separator value between the two leaves (lines 10-11).

After calling *join_leaves. merge_leaf* unlocks the two leaves (lines 4-5 in Figure 3.19)

```
proc join_leaves (left_leaf, right_leaf)
    old_separator := left_leaf.right_sep
    % transfer items from right leaf to left leaf
    left_leaf.right_sep := right_leaf.right_sep
    left_leaf.size := right_leaf.size
    for i = 1 to right_leaf.size do
        left_leaf.k[i] := right_leaf.k[i]
        left_leaf.d[i] := right_leaf.d[i]
    end
    % set rightlinks
    left_leaf.rightlink := right_leaf.rightlink
    right_leaf.rightlink := leaf
    % mark right_leaf as deleted
    right_leaf.marked? := true
    return old_separator
end join_leaves
```

Figure 3.20: *join_leaves(left_leaf, right_leaf)* procedure.

and forks a *complete_merge* operation.

Marking Nodes

When a node n is deleted, we set *n.marked?* to *true*, and *n.rightlink* to point to the node that received n's former contents. This protocol allows ongoing concurrent operations that access a marked node to redirect themselves via rightlinks to the proper node.

Whenever an operation in our algorithm accesses a node, it automatically traverses the node's rightlink if the node has been marked as deleted. This is because:

- The *successor* function return the node's rightlink if the node is marked.
- The *covers?* function return *false* if the node is marked.

Having the *successor* function check for marked nodes ensures that descents from the root to the leaf level in all dictionary operations traverse through rightlinks (e.g., lines 5-14 of procedure *update_descent* in Figure 3.8). The *covers?* check ensures that rightlinks are traversed through marked nodes during the descent phase of dictionary operations where processes "sweep right" along the leaf level (e.g., lines 15-22 of *update_descent* in Figure 3.8), and during the restructuring phase when processes search for a non-leaf node in a given level that covers a separator value (lines 2-7 of procedure *find_parent* in Figure 3.12).

```
proc complete_merge (s, p, stack, l)
    % find and writelock the node in the l'th level which is covers s
    node := findparent(s, stack, l)
    % check if we have to lock right neighbor
    if s = right_sep(node) then
        return two_node_cmerge(node, s, p, stack)
    end
    % else check if s and p are in node.  if not, we have to wait
    index := find_child(node, s, p)
    if index = nil then
        push(stack, node)
        insert <s, p, stack> into node.merge_waiters
        writeunlock(node)
        return
    end
    % check if any waiting operations can be enabled
    start_waiters(s, l)
    % delete s and p from stack
    delete_child(node, index)
    % check if node is an old pointer
    if old_root?(node) then
        fork update_root(node.level - 1)
    end
    writeunlock(node)
end complete_merge
```

Figure 3.21: *complete_merge*$(s, p, stack, l)$ procedure.

3.5.2 Restructuring Phase

The restructuring phase of a *delete* operation begins when the *merge_leaf* procedure forks off a *complete_merge*. Figure 3.21 presents the procedure *complete_merge*$(s, p, stack, l)$. *complete_merge* takes as arguments a separator value s, a downlink to a child p, a stack *stack*, and a tree level l.

complete_merge performs the following tasks. First, it finds the level l node that covers s. Second, it removes s and p from the node. If s is a separator between two level l nodes (thus p is stored in the right neighbor of the node that covers s), then *complete_merge* must treat this case differently from the common case where s is not a separator between nodes. Also, if the level l node that covers s has only one downlink, then *complete_merge* must merge the node and its right neighbor before deleting s and p. Afterwards, to propagate this merge, it invokes a *complete_merge* on the next higher level in the tree.

Finding the Parent Node

Like the procedure *complete_split*, *complete_merge* uses the procedure *find_parent* (shown in Figure 3.12) to find the node in level l that covers s. Recall that the argument *stack* is a stack of node pointers pushed during an *update* operation's descent from the root to the leaf level. *stack* records the last node traversed at each level by the descent.

Earlier in Section 3.4.3, we claimed that the node popped from *stack* might not reach s. While correct for the original Lehman-Yao algorithm and other modified algorithms that do not include a two-phase merge operation [Sag86, LSS87], the assumption that the node must reach s is not true with two-phase merges.

We present the following scenario as an example. Consider the B-link structure in Figure 3.22(a) with a MAX_FANOUT value of 6. Suppose a *delete* operation that deletes key 1 causes the leaves Y and Z to *half_merge*, so the tree now looks like (b). Next, a series of inserts and deletes cause the keys stored in leaf Y to be altered to (c). We assume the *complete_merge* operation (which we have not yet presented) forked by (b)'s *half_merge* has not yet been performed. Suppose the next operation *inserts* 20 into the tree. This operation will cause leaf Y to split, resulting in the structure in (d). The nodes pushed onto the stack by the above *insert* are marked with a checkmark. The *insert* then forks a *complete_split* operation to insert separator 0 and downlink Y' into W, the level 1 node that covers 0. The left separator of X, the node popped from the stack, is greater than 0, so X does not reach 0.

Lanin and Shasha recognized this problem in their algorithm. However, because their B-link tree nodes did not locally store their own left separator values, the check of whether the node popped from the stack reaches s is not so straightforward; they proposed that estimations of a node's left separator value be pushed onto the stack along with the node itself during the descent phase of *update* operations. This estimation is an upper bound of the actual left separator value and can be obtained from the states of previously visited nodes in the descent. Since the estimations are guaranteed to be an upper bound of the actual left separator value, their algorithm is correct. However, this puts additional overhead on *update* operations; the descent phase must compute the estimations as well as push and pop twice as many elements onto and off of the stack. This may be expensive in message-passing architectures. Also, it is possible for the restructuring phase to think the node popped from the stack does not reach s when it actually does. As a result, needless work may be performed.

Our algorithm avoids the problems of Lanin and Shasha's approach. However, it does this by storing and maintaining left separator values on every internal B-link tree node.

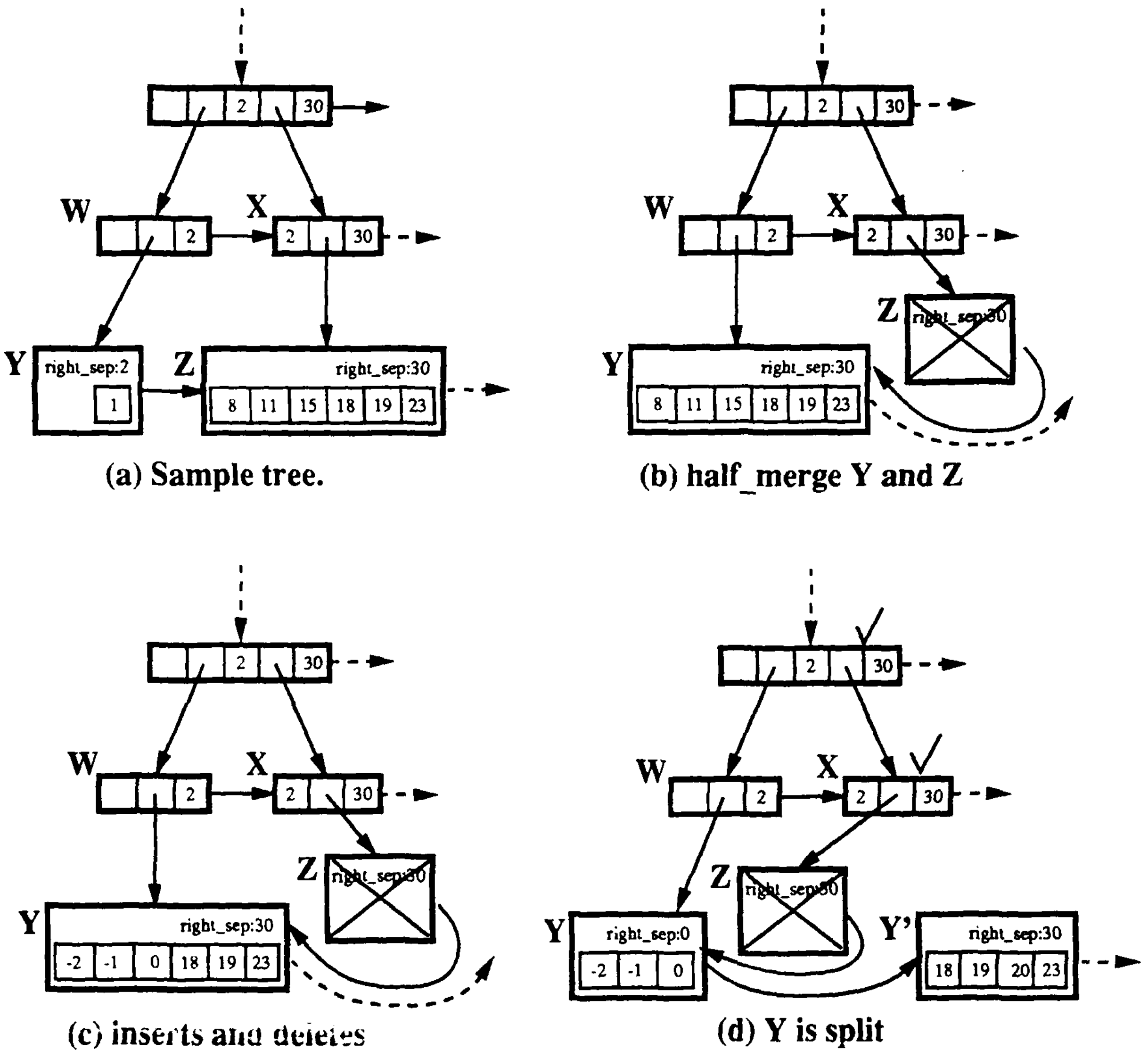


Figure 3.22: Problem with the stack.

Therefore, we can view the differences between our approach and Lanin and Shasha's as a trade-off issue. Our algorithm optimizes the descent phase and the amount of work required to find parents in the restructuring phases of *update* operations at the expense of using extra memory and overhead required to maintain left separator values. The amount of memory and overhead required in maintaining left separator values is minimal; the extra work only occurs when restructuring above the leaf level takes place. In contrast, the overhead required in Lanin and Shasha's method occurs in every *update* operation.

Complete_Merging the Parent

After *complete_merge*$(s, p, stack, l)$ calls *find_parent* to writelock the level l node that covers s (line 1 in Figure 3.21), it checks for the special case where the right neighbor of the node that covers s stores the downlink p (lines 2-4). For example, the *half_merge* performed in Figure 3.22(b) would have forked a *complete_merge* where $s = 2$ and $p = Z$. W covers s, and p is stored in X. If such a case is detected, *complete_split* calls the procedure *two_node_cmerge*. We present this procedure below.

complete_merge then checks if s and p are already stored in the node (line 5). It then performs some synchronization operations (lines 6-12). We ignore them for now, and discuss them in Section 3.6. In that section, we also discuss the case where s and p are not both already stored in the node. The procedure then deletes s and p from the node (line 13). Since both s and p are stored in the node (checked in line 5), s is stored to p's left (also checked in line 5), and s can't be the leftmost separator of the node (or else the node won't cover s), we can safely conclude that the node has more than one child. Thus, we need not merge it. We then call the the procedure *old_root?* to check if the deletion causes the node to be the only node in its level (i.e., a leftmost node with no right neighbor) and to have only one child (line 14-16). If that is the case, then it is possible that the anchor's *root_level field* needs to be updated. Therefore, we fork an *update_root* operation.

Complete_Merge with Two Nodes

two_node_cmerge$(node, s, p, stack)$, shown in Figure 3.23, takes as arguments a node *node* that has already been writelocked, a separator s, a downlink p and a stack *stack*. It assumes $s = right\_sep(node)$ and *stack* is a stack of pointers to nodes whose tree levels occur in consecutive increasing order starting at $node.level + 1$.

two_node_cmerge first writelocks *node's* right neighbor (line 1-2 in Figure 3.23). It then checks if the neighbor's leftmost downlink points to the same node as p (line 3). We already know s is the left separator of the neighbor, because it is the right separator of *node*. We then perform some synchronization operations (lines 4-10) that will be explained in Section 3.6.

To delete both s from *node* and p from *node's* right sibling, we must make a decision in our algorithm. Suppose that given node Y and its right neighbor Z, as shown in Figure 3.24, we must perform a *two_node_cmerge*$(Y, s, p, stack)$. We can either (a) move $p1$ from Y to Z, then delete s and p from Z, or (b) move p from Z to Y, then delete

```
       proc two_node_cmerge (node, s, p, stack)
 1         right_neighbor := node.rightlink
 2         writelock(right_neighbor)
           %check pointer equality. If not equal, then wait
 3         if p != right_neighbor.p[1] || right_neighbor.size = 0 then
 4             push(stack, node)
 5             insert <s, p, stack> into node.merge_waiters
 6             writeunlock(right_neighbor)
 7             writeunlock(node)
 8             return
 9         end
           % check if any waiting operations can be enabled
10         start_waiters(s, node.level)
           % if size of node is 1, then we have to half merge
11         if almost_empty?(node) then
12             return merge_interior(node, right_neighbor, stack)
13         end
           % else we have to shift a pointer to the right and change separator
           % values between the two nodes.
14         right_neighbor.p[1] := node.p[node.size]
15         node.size := node.size - 1
16         right_neighbor.s[0] := node.s[node.size]
           % writeunlock the two nodes
17         new_s := right_sep(node)
18         level := node.level
19         writeunlock(right_neighbor)
20         writeunlock(node)
           % find proper parent and change separator
21         new_stack := stack_copy(stack)
22         complete_merge(old_s, right_neighbor, new_stack, level + 1)
23         complete_split(new_s, right_neighbor, stack, level + 1)
24     end two_node_cmerge
```

Figure 3.23: *two_node_cmerge*(*node*, s, p, *stack*) procedure.

s and p from Y. It turns out choice (b) is incorrect. If some ongoing process with key argument k, where $s < k \leq s2$, visits parent X, it will decide Z is the next node to visit. But Z no longer reaches k. Therefore, we must choose (a).

two_node_cmerge first checks if *node* has only one child (lines 11-13 in Figure 3.23). If that is the case, then we must perform a *half_merge* operation on *node* and its neighbor. (Shifting its only downlink to its neighbor would cause *node* to be empty.) The procedure *merge_interior* performs this operation; we discuss it below.

If *node* has more than one child, *two_node_cmerge* transfers *node*'s rightmost downlink to *node*'s neighbor, and then deletes s and p, as shown in Figure 3.24(a). Specifically, *two_node_cmerge* removes s and *node*'s rightmost downlink from *node* by decrementing

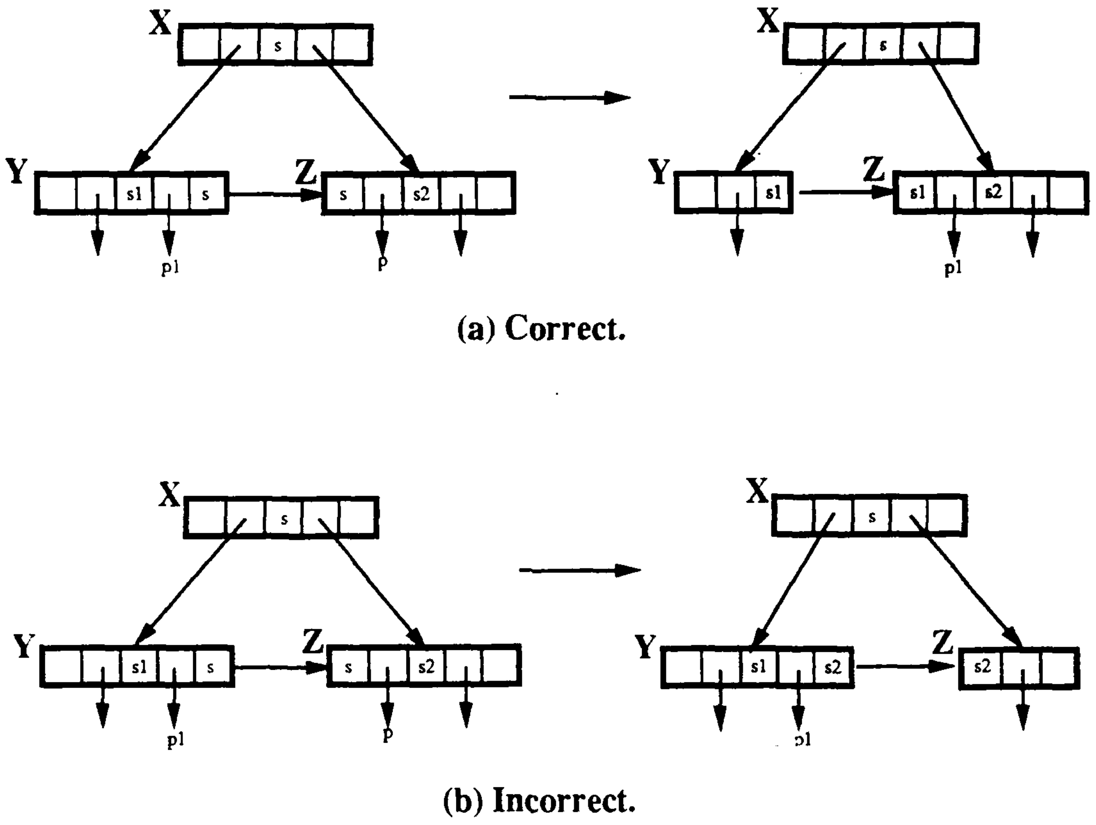


Figure 3.24: Implementation choice for *two_node_cmerge*.

node.size, replaces *p* in *node's* right neighbor with the *node's* former rightmost downlink, and updates the left separator of *node's* neighbor (lines 14-16).

The approach we present for this special case of *complete_merge* is similar to the approach taken by Lanin and Shasha [LS86]. They propose first merging the two nodes. Then they remove the downlink and separator from the resulting node. If the node is over-utilized, they split the node in two. The advantage of our approach is that we do not merge any nodes, and we shift only one downlink from the left to the right node. Thus we hold writelocks for a much shorter length of time (merging two nodes requires much more work than swinging a downlink); we also do not needlessly delete nodes (which saves memory).

two_node_cmerge has now completed the deletion of *s* and *p*. However, the deletion decreased the separator between *node* and *node's* right neighbor. In Figure 3.24(a), the separator has changed from *s* to *s*1. Note that because $s1 < s$, the tree can still support dictionary operations. Any dictionary operation that traverses the wrong node as the

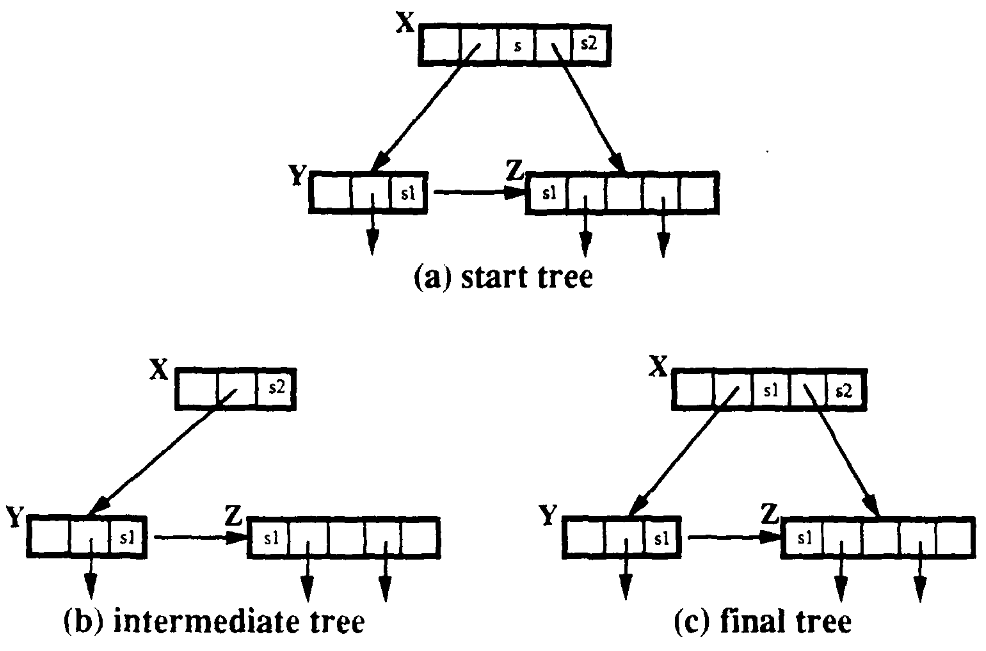


Figure 3.25: Changing separator values.

result of the discrepancy between $s1$ (the actual separator between the two nodes) and s (the stored separator in the parent) can redirect itself to the proper node by using rightlinks. However, to maintain efficiency, s must be updated to $s1$ in X. To do this, *two_node_cmerge*(*node*, s, p, *stack*) performs two operations. The first is a *complete_merge* to remove the separator s and the downlink to *node*'s right neighbor from the parent (line 21-22 of Figure 3.23). The second is a *complete_split* to insert the separator $s1$ and a downlink to *node*'s right neighbor to the parent (line 23).[5] Note that in line 21, the procedure *stack_copy* creates a new stack whose contents are the same pointers as *stack's*. We need a separate copy since both operations will use *stack's* pointers.

The correctness of the above two operations should be obvious. Figure 3.25(a) shows a parent X with downlinks to Y and Z after *two_node_cmerge* changes the separator value between Y and Z from s to $s1$ (where $s1 < s$). Part (b) shows the state of the parent X after the completion of the *complete_merge*. This tree structure can still support dictionary operations. The final state of X after performing the *complete_split* is shown

[5]A more efficient implementation would be to define a procedure that performed the tasks of the above two operations, i.e., remove s and the downlink to s's right and re-insert the downlink with separator $s1$. In the common case where s and $s1$ are both covered by the parent, such a procedure would avoid re-writelocking the parent when the downlink is re-inserted.

```
    proc merge_interior(left_node, right_node, stack)
1       old_separator := join_interior(left_node, right_node)
        % check if left_node is now a old root
2       if old_root?(left_node) then
3           fork update_root(left_node.level - 1)
4       end
        % unlock nodes
5       l := left_node.level + 1
6       writeunlock(right_node)
7       writeunlock(left_node)
        % complete_merge
8       complete_merge(old_seperator, right_node, stack, l)
9   end merge_interior
```

Figure 3.26: *merge_interior(left_node, right_node)* procedure.

in part (c). The result is that X updates its separator s to $s1$.

Half_Merging Interior Nodes

complete_merge calls the procedure *merge_interior(left_node, right_node)*, shown in Figure 3.26, to *half_merge* interior nodes. It takes as arguments two interior neighbors. It deletes the separator between the two nodes as well as the leftmost downlink in *right_node*, and then merges the nodes. It assumes the two nodes to be merged are already write-locked. *merge_interior* first calls the procedure *join_interior* to merge the contents of the two nodes (line 1 in Figure 3.26).

join_interior(left_node, right_node), shown in Figure 3.27, merges neighbors *left_node* and *right_node* and returns the old separator value between them. It first saves the old separator (line 1 in Figure 3.27). Then it moves all the data from *right_node* to *left_node*, except for the leftmost downlink and separator (lines 2-7), which are discarded. *join_interior* then calls *join_waiters* which updates the *merge_waiters* and *split_waiters* fields (line 8). We explain this operation in Section 3.6. After setting the rightlinks properly (lines 9-10) and marking the right node as deleted (line 11), the old separator value is returned (line 12).

After calling *join_interior*, *merge_interior* checks if the merge has caused the tree to shrink levels (lines 2-4). This may happen if *left_node* and *right_node* were leftmost and rightmost nodes in their tree level, respectively, and each of them had only one downlink. If this is the case, *merge_interior* forks off the procedure *update_root*. Finally, *merge_interior* releases its two locks, and invokes *complete_merge*.

```
proc join_interior(left_node, right_node)
    % move data from right_node to left_node
    old_separator := right_node.s[0]
    left_node.size := right_node.size
    left_nodes[1] := right_node.s[1]
    for i = 2 to right_node.size do
        left_node.s[i] := right_node.s[i]
        left_node.p[i] := right_node.p[i]
    end
    % concatenate waiter lists
    join_waiters(left_node, right_node)
    % set rightlinks
    left_node.rightlink := right_node.rightlink
    right_node.rightlink := left_node
    % mark right_neighbor as deleted
    right_node.marked? := true
    return old_separator
end join_interior
```

Figure 3.27: *join_interior(left_node, right_node)* procedure.

3.6 Coordinating Background Processes

Our discussion of the operations assumed that background *complete_splits* and *complete_merges* can execute independently without synchronization. Sagiv [Sag86] points out that the original Lehman-Yao algorithm can perform independent *complete_split* operations without extra synchronization. However, the original Lehman-Yao algorithm did not provide for two-phase merges.

The problem is that with two-phase merges, *complete_merge* and *complete_split* operations must synchronize with each other if their separator arguments are equal. A *complete_merge* that deletes separator s and downlink p from a node must wait for the *complete_split* that originally inserted separator s and downlink p. *Complete_splits* must make sure the separator they are inserting does not already exist in the node.

3.6.1 Examples of the Problem

Consider the following example, where node X has a child Y, which is *half_split* into Y and Z, by an *insert*. The separator between Y and Z is s. The *half_split* forks an independent process to perform a *complete_split* that will add to parent X the separator s and a downlink to Z. Later, a *delete* *half_merges* Y and Z, and marks Z as deleted. The *half_merge* forks an independent process to perform a *complete_merge* operation that

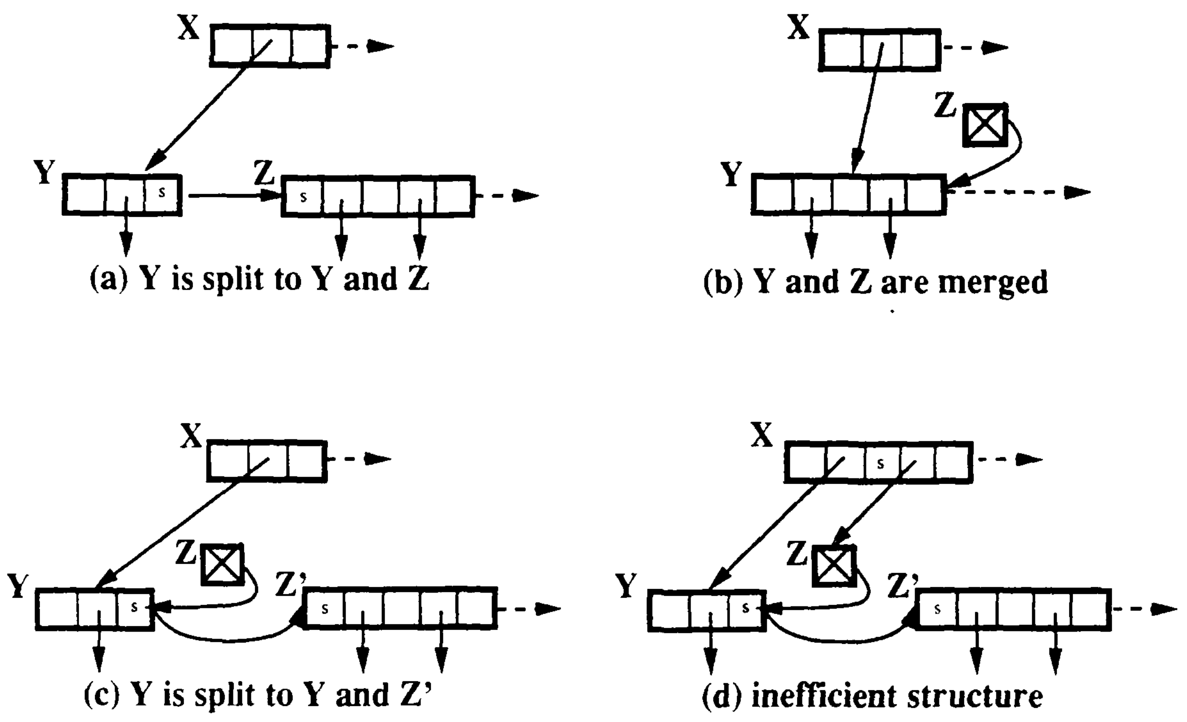

X
Y
Z
s
(a) Y is split to Y and Z
X
Z
Y
(b) Y and Z are merged
X
Z
Y
Z'
s
(c) Y is split to Y and Z'
X
s
Z
Y
Z'
s
(d) inefficient structure

Figure 3.28: Synchronization example.

will remove the separator s and downlink to Z from X. If the *complete_merge* operation is performed before the *complete_split* adds s and Z to X, there is a problem since the downlink and separator that the *complete_merge* tries to delete have not yet been inserted. An similar example is where a *complete_split* tries to insert a separator and downlink into a node, when the same separator value already exists. (This could happen if a *complete_merge* operation that will remove the separator value has not yet executed.)

A more complex example is shown in Figure 3.28. In (a), node Y has just been split into Y and Z. Later in (b), separator s and Z's leftmost downlink are removed, and Y and Z are merged. Even later in (c), a downlink is inserted in Y, causing it to be split into Y and Z', with separator s between Y and Z'. The order in which we perform the three background operations forked by the above is important.

If the *complete_split* that inserts s and Z' occurs first, followed by the *complete_merge* that removes s and Z, we may have a problem. If the *complete_merge* operation does not check downlinks along with separator values, we may inadvertently delete Z' from the tree. Later, the *complete_split* that inserts s and Z may reach X and insert s and Z into X, resulting in the structure shown in (d). While this tree can still support dictionary operations, it is obviously not the most efficient structure.

3.6.2 Solution

The obvious solution to the example shown in Figure 3.28 is to check both downlinks and separators already stored in the node before updating the node. This is done in line 2 in the *complete_split* procedure (Figure 3.11), lines 2 and 5 in *complete_merge* (Figure 3.21), and line 3 in *two_node_cmerge* (Figure 3.23). The checks in *complete_split* check if a separator value already exists in the node. If this is the case, then the *complete_split* cannot continue. The checks in the *complete_merge* procedures check if the separator to be deleted exists in the node, and whether the downlink to the separator's right matches the downlink to be deleted. If the check fails, then the *complete_merge* cannot continue. Notice that we have not solved the problem entirely. A *complete_merge* or *complete_split* operation that cannot continue must somehow restart at a later time.

One simple solution is to "spin" if separator and downlink checks are not satisfied. This would mean releasing all writelocks and recursively calling the same *complete_split* or *complete_merge* procedure with the same arguments if the checks fail. Lanin and Shasha [LS86] provide such a solution for their algorithm. Unfortunately, this is not correct. This solution solves the problem posed by the first example above; the *complete_merge* procedure will spin until the *complete_split* operation inserts the separator and downlink it wants to delete. However, spinning will not solve the problem in the example shown in Figure 3.28. If the first *complete_split* inserts into X the separator s and the downlink to Z' to its immediate right, the *complete_merge* that tries to delete s and Z will spin, because $Z \neq Z'$. Since separator value s already appears in X, the second *complete_split* that tries to insert s and Z will also spin. These two procedures will spin as long as the separator s and downlink Z' stay in X. It is possible that these two values will never be deleted from X. Thus the two procedures could spin forever.[6]

Instead of spinwaiting, we provide two lists in every non-leaf n: *n.split_waiters* and *n.merge_waiters*. When a *complete_split* or *complete_merge* fails its downlink or separator check, it first inserts into the appropriate list (*n.split_waiters* for *complete_splits* and *n.merge_waiters* for *complete_merges*) enough information to restart itself, and then releases its writelocks and terminates. The restart information consists of the separa-

[6]Cases where the "spinwait" solution fails happen very rarely. Such a scenario requires a large number of updates occurring in a short period of time causing nodes to be split, merged, and split again along the same separator value. For many applications, it may be adequate to implement the spinwait solution and treat occurrences of the above case as an error.

```
 proc start_waiters (s, l)
      % check if any waiting operations can be restarted
      for each triple <s1, p, stack> in node.split_waiters do
          if s = s1 then
              fork complete_split(s1, p, stack, l)
              remove <s1, p, stack> from node.split_waiters
            end
      end
      for each triple <s1, p, stack> in node.merge_waiters do
          if s = s1 then
              fork complete_merge(s1, p, stack, l)
              remove <s1, p, stack> from node.merge_waiters
          end
      end
end start_waiters
```

Figure 3.29: *start_waiters*(s, l) procedure.

tor, downlink, and stack arguments of the *complete_merge* or *complete_split*. The list insertions occur in lines 2-7 of the pseudocode for *complete_split* (Figure 3.11), lines 5-11 for *complete_merge* (Figure 3.21), and lines 3-9 for *two_node_cmerge* (Figure 3.23). For most applications, we expect the lengths of these lists to be small; insertions into these lists require scenarios where large number of localized updates occur in a short period of time, causing nodes to split, merge, and split again along the same separator value. The probability of such occurrences is small.

All incoming *complete_splits* and *complete_merges* must check the lists to find any "waiting" operations that they can enable. They do this by calling the procedure *start_waiters*, right after they make their downlink and separator checks. Figure 3.29 presents the pseudocode for *start_waiters*. This procedure forks off a *complete_merge* and *complete_split* invocation for each element stored in the two lists whose separator value is equal to the argument s.

Using the lists in the manner described above does not quite solve the problem of illustrated by the example shown in Figure 3.28. Instead of "spinning forever" (as in Lanin and Shasha's algorithm), the two lists will contain elements that will never be removed. While the algorithm is correct, the extra elements introduce unnecessary overhead. A simple way to avoid this problem is to remove pairs elements in the lists that can "cancel each other" (i.e., an element on each of the lists whose values are identical) every time an element is inserted into one of the lists.

Note that *half_merge* and *half_split* operations must properly divide and merge these lists. *divide_interior* does this by calling the function *divide_waiters*(*node*, *new_node*)

```
proc divide_waiters (node, new_node)
    for each <s, p, stack> in node.split_waiters do
        if s > node.s[node.size] then
            transfer <s, p, stack> from node.split_waiters onto
                new_node.split_waiters
        end
    end
    for each <s, p, stack> in node.merge_waiters do
        if s > node.s[node.size] then
            transfer <s, p, stack> from node.merge_waiters onto
                new_node.merge_waiters
        end
    end
end divide_waiters
```

```
proc join_waiters(left_node, right_node)
    append right_node.split_waiters onto left_node.split_waiters
    append right_node.merge_waiters onto left_node.merge_waiters
end join_waiters
```

Figure 3.30: Functions that divide and join wait lists.

(line 9 of Figure 3.15); *join_interior* calls the function *join_waiters*(*left_node*, *right_node*) (line 8 in Figure 3.27). Figure 3.30 displays both functions.

3.7 Parent Pointers

In this section, we present the concept of *parent pointers*, an idea proposed by Eric Brewer. Each node is augmented with a pointer to its parent, or at least to a node to the left of its parent. The addition of such pointers may significantly reduce the overhead resulting from *update* operations.

Whenever an *insert* or *delete* operation performs its descent phase, a stack must be maintained to record the last node visited at each level of the tree. For message-passing architectures, stack operations may be quite expensive. The algorithm presented by Lanin and Shasha requires even more overhead because estimations of left separator values are pushed onto the stack as well. The stack is used only if the *update* operation requires restructuring above the leaf level. (I.e., it is rarely used.)

If each node in the B-link tree maintains a pointer to its parent, then this generally unnecessary overhead can be avoided. Rather than popping nodes from the stack (and in the case of Lanin and Shasha's algorithm, popping estimations of left separator values), the restructuring phase of *update* operations will follow parent pointers.

Unfortunately, the addition of such pointers requires extra maintenance. For example, every time nodes are split or merged, parent pointers of the affected children must be updated as well. However, this can be done in a lazy fashion by background processes in an approach similar to the two-phase approach for merges and splits. We can view the lazy update to parent pointers as a "third phase."

Consider the simple example in Figure 3.31. In part (a), we see an example tree with parent pointers. Node X has become full and needs to be split. A *half_split* occurs in (b). Note that the parent pointers of the children of both X and Y point to X. This is acceptable since Y can be reached from X through a rightlink. In (c), the corresponding *complete_split* updates the parent of X. Finally in (d), the parent pointers of the children of X and Y are lazily updated. The operations in (c) and (d) can be performed concurrently. Merges can be handled in a similar approach.

The strategy in the example in Figure 3.31 guarantees an invariant in the B-link tree structure, *in which the actual parent of a node n can be reached through zero or more* rightlinks from the node pointed to by n's parent pointer. With minor modifications, we can easily build a variation of the algorithm presented in this chapter that uses parent pointers and maintains this invariant.[7] We reserve this work for future studies.

3.8 Summary

In this chapter, we presented a concurrent B-link tree algorithm based on the Lehman-Yao algorithm [LY81] with modifications by Sagiv [Sag86]. The algorithm has the following important properties:

- The descent phase of every dictionary operation locks only one node at a time.
- *Process overtaking* can cause descents to stray to the left of the proper path, but *rightlinks* allow for redirection to the proper nodes.
- The restructuring phases for *inserts* and *deletes* use a two-phase strategy that allows much of the restructuring to be completed in the background. The *insert* restructuring phase locks one node at a time; the *delete* restructuring phase (based on ideas by Lanin and Shasha [LS86]) locks at most two nodes concurrently.

[7] Without the invariant, parent pointers might direct restructuring phases to nodes that cannot reach the actual parent. In such cases, we can use a strategy that either accesses the leftmost node in the level or performs a descent from the root to find a node that can reach the parent.

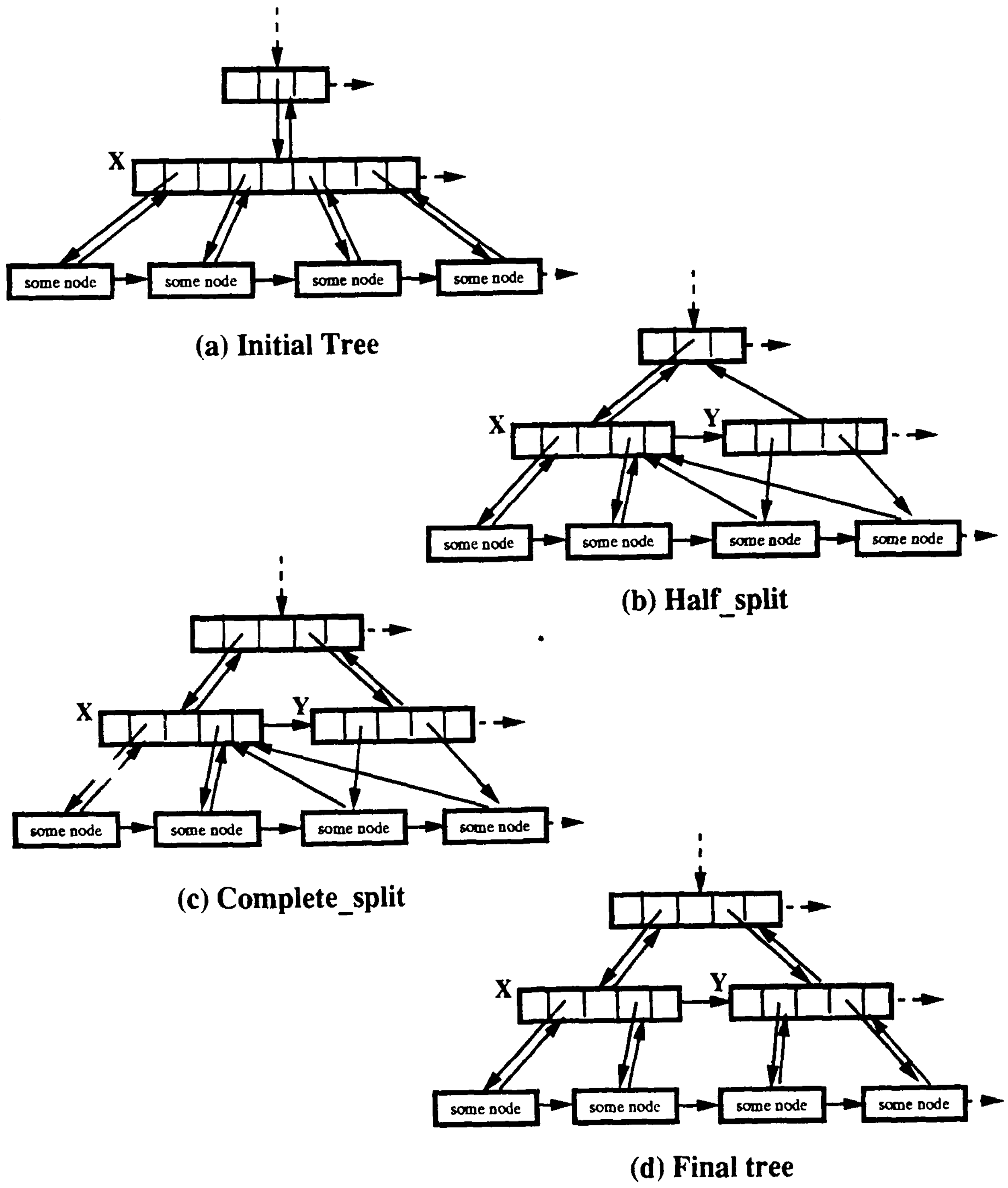


Figure 3.31: Parent pointers.

- The anchor's *root_level* field is maintained by a *critic* process that is created on demand and removed when its tasks are completed.

We finally show how B-link tree nodes can be augmented with parent pointers to remove the overhead required in maintaining stacks during the descent phase of *update* operations.

Chapter 4

The Multi-Version Memory Algorithm

This chapter describes another new concurrent B-tree algorithm. The algorithm is designed to work well in large-scale parallel or distributed systems in which the number of processors sharing the tree is large, or the communication delay between processors (or between processors and global memory in a shared memory architecture) is large compared to the speed of local computation.

In an application that uses a concurrent B-tree, replication schemes such as caching are likely to be important tools for achieving high performance. For example, every dictionary operation visits the root of the tree. The probability that an operation will update the root is small. If no replication is used, resource contention for the system component that stores the root will likely become the limiting factor in performance. Replication improves performance in part by allowing processes to access data in local memory, thus avoiding the delay involved in accessing a remote memory, and in part by replicating data so that many processes can read it in parallel.

Most replication schemes guarantee the memory to be *coherent*, which constrains the states of the replicated copies so that read and write operations appear to be atomic. These constraints require significant synchronization between readers and writers, and also require communication to update or invalidate copies after a processor has written to memory. We call all such replication schemes *coherent shared memory.* Archibald and Baer [AB86] present an analysis of a number of such schemes.

The basis of the B-tree algorithm we describe in this chapter is an abstraction that is similar to coherent shared memory, but provides a weaker semantics; we call this abstraction *multi-version memory* [WW90]. Multi-version memory uses replication, providing the advantages described above, but weakens the semantics of coherent shared memory by allowing a process reading data to be given an old version of the data. For example, if we are using hardware caches, a process may simply use its local cache's copy, even if the copy has not yet recorded recent updates by other processes. While this semantics is not as generally useful as that provided by coherent shared memory, it turns out to be adequate for the B-tree algorithm presented in this chapter.

The advantage of the weaker semantics provided by multi-version memory is that it can be implemented more efficiently than coherent shared memory. The implementations of multi-version memory that we describe below have several important characteristics:

- They allow processes reading data to run in parallel with a process writing the data.
- They eliminate "cache misses" resulting from invalidation caused by writes by other processes.
- They eliminate the need for processes to wait for messages that update or invalidate replicated copies.

The net result of these characteristics should be higher throughput and lower latency of B-tree operations. For example, by allowing processes reading data to run in parallel with a process writing the data, we eliminate the need for a descending process to block while an update propagates up the tree. By allowing a process to use an old version of a B-tree node in a replicated copy even after another process has updated the node, we avoid the need to wait for communication required to bring the copy up to date. As presented in Chapter 5, our experiments show that the performance improvement obtained by using multi-version memory for non-leaf nodes is substantial in a large-scale system with many processors, or in which communication is expensive.

If replication is provided by hardware caches, implementing multi-version memory may require managing these caches in software. Others have proposed software cache management as a way of tailoring the cache management algorithm to the needs of the application [BMW85, SS88, CSB86, BCZ90]; however, they all provide coherent shared memory (defined by either linearizability [HW90] or sequential consistency [Lam79]), and optimize the implementation to take advantages of characteristics of the application. The programmer still sees reads and writes as atomic operations.

Multi-version memory goes one step further; in addition to tailoring the cache management *algorithm* to the needs of the application, we also tailor the *semantics* of the memory. This suggests that it could be fruitful to view cache management as an application-level replication problem, where the user can specify as part of the application both the semantics of the shared data and the algorithm used to manage caches. Such an approach fits naturally into an object-oriented programming style based on inventing application-specific abstract data types, such as that advocated by Liskov and Guttag [LG86]. A multi-version memory object is simply an instance of an abstract data type, whose specification gives a different semantics to read and write operations than does the specification of coherent shared memory. Also, the user can encapsulate complex cache management algorithms in the implementations of the abstract data types, and can change the management scheme depending on the access patterns of the application. Cheriton [Che86] has made a similar suggestion, and has given examples of how weak notions of consistency can be useful in distributed systems. Here, we apply the general idea to parallel data structures.

We base the general mechanics of our new B-tree algorithm on the mechanics of the coherent shared memory algorithm presented in the previous chapter. There have not been many studies investigating the performance of concurrent B-tree algorithms (e.g., [BS77, MR85, KW82]); however, the studies that have been done (based on both simulations and analytical models) show that the Lehman-Yao algorithm, on which the algorithm in Chapter 3 is based, should perform better than any other algorithm designed to date [JS90, LSS87].

Instead of describing a single algorithm, we present our algorithm as a transformation of any B-tree algorithm that uses coherent shared memory for all nodes and that satisfies some additional assumptions. This allows our technique to be applied to different link method algorithms. For example, recall the discussion in the previous chapter about how Lehman-Yao based B-link tree algorithms can implement restructuring phases for *deletes*. Sagiv [Sag86] proposes one method, Lanin and Shasha [LS86] propose another, and we describe a third. Our transformation can be applied to any of these algorithms.

We structure the remainder of this chapter as follows. First, in Section 4.1, we give a specification of multi-version memory, and discuss how it can be implemented. Then in Section 4.2, we describe our transformation. In Section 4.3, we describe how the transformation can be applied to the coherent shared memory algorithm presented in the previous chapter. We conclude in Section 4.4 with a summary of the chapter.

4.1 Multi-Version Memory Schemes

In this section, we present multi-version memory replication schemes. We begin by describing the operations provided by a multi-version memory object as seen by a client of the abstraction. Then we present an implementation of multi-version memory that is architecture-independent. We conclude with a discussion of how multi-version memory can be implemented on existing architectures.

4.1.1 Specification

In a multi-version memory, the abstract state of an object consists of a sequence of versions, plus an exclusive lock used to synchronize writers. The first version in the sequence is the *initial* version, and the last version is the *current* version. Writers update the object by extending the sequence of versions with new versions (thus changing the current version), and readers read the object by choosing and reading some version. The specification allows the reader to read *any* version, not just the current version (which is what coherent shared memory would require). As discussed in the next section, this nondeterminism allows us to implement a multi-version memory so that readers can run in parallel with writers. Performance of applications that can use multi-version memory will probably be better if readers obtain and read recent versions, but the specification of a multi-version memory requires that the application be prepared for its readers to obtain an arbitrary version.

As discussed in Chapter 2, a coherent shared memory provides operations to read and write memory, as well as additional operations for synchronization (e.g., operations on exclusive locks or read/write locks). Such synchronization operations are applied to ***lock objects*** that are typically separate from shared data; the association between a lock and the data it protects is typically just a program convention. An object should be read only when its associated lock object is locked for reading, and written only when its associated lock object is locked for writing.

A multi-version memory also provides operations that read and write memory, as well as synchronization operations. However, the synchronization operations are more closely coupled with the read and write operations than in a conventional shared memory. A multi-version memory provides seven operations: *read*, *write*, *readpin*, *readcurrent*, *readunpin*, *writepin* and *writeunpin*. (We use the term "pin" for reasons that should become clear below: "pinning" is to a multi-version memory what "locking" is to a coherent shared memory.) In specifying the operations, we view each as an atomic action;

the implementation must guarantee that the apparent behavior is as if the operations execute atomically in an order consistent with their real-time order. This property is called *linearizability* [HW90].

A process reads an object by issuing a *readpin* or *readcurrent* operation, then issuing some *read* operations, and finally issuing a *readunpin* operation. The *readpin* operation has the effect of selecting an arbitrary version of the object; the *readcurrent* operation always selects the current version. The *readunpin* operation simply informs the system that the process is done with the version of the object selected by the previous *readpin* or *readcurrent* operation; it is needed for performance, but has no observable effect on the state of the object. A *read* operation uses the version selected by the immediately preceding *readpin* or *readcurrent* operation. A process can issue a *read* only if the process has a version selected; a *read* is illegal if no version has been selected since the latest *readunpin*.

A process writes an object by first issuing a *writepin* operation, then issuing some *read* and *write* operations, and finally issuing a *writeunpin* operation. The *writepin* operation has the effect of first obtaining an exclusive lock on the object (blocking if some other process holds the lock), and then copying the current version of the object into a private version local to the process. This copy is then treated as the process's selected version during its subsequent *read* and *write* operations. The *writeunpin* operation updates the object by appending the process's private version to the object's sequence of versions (thus creating a new current version), and then releasing the lock. A process can issue a *write* operation only if it has the object pinned for writing.

The exclusive lock acquired by the *writepin* operation has the effect of sequencing writers so that each *write* sees the effects of all previous *writes*. However, the lock has no effect on a reader. As discussed in the next section, we can implement a multi-version memory so that a reader can read an object while a writer has it pinned for writing. Readers and writers still need some low-level synchronization, but the delays involved can be made quite short. In addition, the delays incurred by a reader do not depend on how long a writer keeps an object pinned for writing.

As with ordinary shared memory, multi-version memory sequences the operations of writers. Thus, as long as each *writepin* block preserves consistency (each "block" consisting of the *read* and *write* operations between a *writepin* and the subsequent *writeunpin*) and the initial version is consistent, every version of the object is consistent. This means that a reader sees a consistent state of an object each time it reads the object. However, a reader can see an *old* version, and the specification allows it to be arbitrarily old.[1]

4.1.2 Implementations

A variety of MIMD architectures can implement multi-version memory, including shared memory style machines and message-passing multiprocessors, as well as distributed systems. We describe one such implementation and some variations below in a way that is reasonably independent of the particular architecture. We first explain how to represent a multi-version memory object, and then we describe the implementation of each of the operations.

Our implementation represents each multi-version memory object by a (single) base copy that contains the current version of the object, a mutex to serialize write operations, and some replicated copies. For example, if the multi-version memory is implemented directly in the hardware caches of a shared memory architecture, the base copy resides in shared memory while the replicated copies refer to the cached copies of the object. Each replicated copy contains a flag that indicates whether the copy is *pinned.* The implementation can discard an unpinned copy to free up space for some other use, or replace it with a more current copy to improve the application's performance; however, it cannot discard or replace a pinned copy. We assume that each of the processes, at a given time, will use either zero or one replicated copies of the object. We also assume that copying the contents of the base copy to or from a replicated copy is an atomic action.

An operation that reads or writes the object issued by a process uses a replicated copy "assigned" to the process. If the multi-version memory is implemented within hardware caches, the assigned copy may refer to the process's cached copy. On the other hand, a distributed system implementing multi-version memory in software may have a fixed number of replicated copies of an object, so the system may randomly assign an unpinned replicated copy to the process. A process reading data must first issue a *readpin* or *readcurrent* on the data; a process writing data must issue a *writepin* on the data. The implementation then ensures that a process will always be assigned a replicated copy when it issues a read or write operation.

A *readpin* operation first assigns an *unpinned* replicated copy to the process. This

[1] We could add additional constraints to the specification. For example, we might require *read_pin* to choose a version that is no older than any other version already used by the process. Alternatively, we could require it to choose one of the k most recent versions. The B-tree algorithm presented in this chapter does not need such constraints, so we will not discuss them further.

may require creating a new replicated copy and copying the base copy's contents into the new copy (e.g., a multi-version memory is implemented in a hardware cache, and a process does not have the shared object in its cache), or it may require blocking the process until an *unpinned* copy becomes available (e.g., a multi-version memory is implemented in a distributed system with a fixed number of replicated copies for each object, and all the copies are pinned). It then marks the replicated copy as *pinned.* The *readpin* performs the copy assignment and marking in one atomic step. A *readcurrent* operation is similar, except that it ensures that the process's replicated copy is current (i.e., is equal to the base copy) by copying the base copy into the process's new copy, if necessary. (We discuss below how to ensure that a replicated copy is current.) A *readunpin* operation simply marks the process's copy as *unpinned.*

A *writepin* operation first acquires the mutex in the representation of the object. By using the same protocol as for *readpins*, an *unpinned* replicated copy of the object is assigned to the process. It ensures that the replicated copy is current, and then marks the copy as *pinned.* Like the *readpin* operation, a *writepin* performs the copy assignment and marking in one atomic step.

There are several ways to ensure that a writer is assigned a current replicated copy during a *writepin* operation. (The implementation for the *readcurrent* operation can use similar techniques.) The simplest is just to copy the base copy into the writer's replicated copy as part of the *writepin* operation. However, if the writer's copy is already current, this approach incurs avoidable overhead. Another approach keeps version numbers with each copy: the version number is incremented at each *writeunpin* operation, and can be used to tell if a replicated copy is obsolete. However, this takes extra space, and there is no theoretical bound on the number of bits needed for the version number. In addition, the *writepin* operation still needs to retrieve the version number of the base copy, which could involve a significant communication cost. A third way is for all write operations to "invalidate" all replicated copies of the object. (For such schemes, we associate with each replicated copy a *valid* flag.) That way, if a *readcurrent* or *writepin* is assigned a replicated copy marked invalid, it knows it has to copy the contents of the base copy.

In one atomic step, a *writeunpin* operation copies the process's replicated copy back into the base copy,[2] marks the replicated copy as *unpinned*, and releases the mutex. (The copy needs to be done only if the process has modified the replicated copy since its last issued *writepin* operation.)

[2]Alternatively, an "ownership" cache-coherence protocol can avoid copying back to the base copy right away.

Although the specification of multi-version memory allows the version selected by a *readpin* to be arbitrarily old, the performance of many applications using multi-version memory will likely be better if selected versions are as close to the current one as possible. Thus it is desirable to have *writeunpin* operations propagate to all replicated copies the changes made to the base copy. Notice that a *writeunpin* does not have to wait for this propagation to finish in order to release the writer's mutex and complete. Instead, the system can create a "background process" after the *writeunpin* finishes to broadcast the base copy changes to all the replicated copies.

There are several ways to propagate changes to replicated copies. One way is to invalidate the replicated copies. In a multi-version memory implemented on hardware caches, we can simply remove the cached objects from the processes' caches. However, we can only remove unpinned cached copies. (Otherwise a process performing a read operation might not have an assigned copy anymore. If the base copy were then used to create a new copy, the process might read two different versions of the object between a single *readpin* operation and the subsequent *readunpin* operation.) We can satisfy this constraint by associating an additional *valid* flag with each cached copy indicating whether the copy is "current" or "obsolete," and invalidating a pinned cached copy by marking it as obsolete. A *readunpin* operation can then check whether the process's cached copy is obsolete, and if so, remove it from the cache.

Alternatively, invalidation could simply mark all replicated copies as obsolete, regardless of whether the copy is pinned. A copy marked obsolete can be brought up to date at some convenient time, but the processes using the copy need not be delayed while this happens.

Instead of invalidating, we can also directly update an obsolete replicated copy with the base copy. As with invalidations, an important constraint is that a replicated copy can be updated only if it is not pinned. We satisfy this constraint by queueing update requests for pinned copies, so that the next *readunpin* or *readpin* can update the replicated copy with the new value. (In fact, we only need to queue the latest update request for each copy.) Alternatively, an update request for a pinned copy could mark the copy as obsolete; the next *readunpin* could then copy the base copy into the replicated copy.

4.1.3 Multi-Version Memory and Existing Architectures

The implementations of a multi-version memory object described above have several key characteristics:

- By using an old version, a process reading the object can proceed while another process is writing to the object.

- By allowing lazy propagation of an update from the base copy to replicated copies, we spread over time the invalidation load for a heavily shared object; this should result in a more even load on the network, and help avoid the kinds of saturation problems discussed by Pfister and Norton [PN85].

- The only constraint on when to update a replicated copy is that the copy cannot be pinned during the update. In particular, there are no requirements that a replicated copy be updated by a certain time.[3] Thus, one possible implementation allows readers to avoid waiting whenever possible. For example, if a *readpin* operation finds the process's replicated copy marked obsolete, it might make sense to use the old copy if there would be a long delay in getting the base copy.

For applications that can use a multi-version memory, the above three characteristics can significantly improve performance by increasing concurrency and throughput, and decreasing latency.

Our description of the implementations above can be applied to both shared-memory and message-passing architectures. On a shared-memory architecture, we might implement multi-version memory in hardware caches and create replicated copies dynamically when needed. On a message-passing architecture, such as the J-machine [DCF$^+$89], we might instead maintain in software a fixed number of copies on different processors, which serve to spread out the load and reduce contention. (Dally's "distributed objects" [DC88] or Chien's "concurrent aggregates" [CD90, Chi90] might be useful substrates for implementing a multi-version memory on a message-passing machine.)

The software management approach can be implemented on existing architectures. However, updating the replicated copies in software involves substantial overhead. Thus supporting multi-version memory in the hardware level seems like an attractive alternative. Hardware support for fast block copy would be useful. Hardware caches also provide fast associative lookup.

Using hardware caches has two further potential advantages. First, the dynamic replication may adapt better to changes in load, since the number of copies that exists

[3] Afek, et al. describe a "lazy" cache algorithm that ensures sequential consistency [ABM89]; it imposes relatively weak constraints on propagation of updates from one cache to another, but still requires a processor to wait at certain times until updates have been propagated to other caches (or the input queues at other caches). Multi-version memory imposes essentially no constraints; perhaps its implementation should be called a "lazier" cache algorithm.

depends on how many processes are actively using the object. Second, the use of local cache memories may allow faster access, since in a software-based multi-version memory implementation on a message-passing machine, a process desiring access to an object must send a message to a processor holding a copy of the object.

However, the shared-memory approach also incurs some overhead in creating and deleting copies dynamically. Furthermore, we know of no architectures that allow an object to be "pinned" in a cache. Supporting this raises the obvious problem of what to do when the cache is full with pinned objects and something needs to be removed to make room for another object; since in the B-tree algorithm described below, a process never pins more than a small constant number of objects (at most two) at a time, one reasonable choice might be to treat it as an error.

There are other potential problems with supporting multi-version memory in hardware. A hardware cache typically imposes a fixed size on cached objects. Forcing the programmer to break a large object into several small ones will not work, since there is no way to guarantee that the versions of the different small objects read by a process are consistent. Supporting variable-sized objects in hardware caches is difficult. A reasonable compromise might be the approach taken in the VMP system [CSB86], which uses a large cache page size. This might be adequate for many applications. (VMP handles cache coherence in software, which at least gives the potential of implementing multi-version memory to take advantage of the fast block copy and associative access provided by the hardware, but it is not clear whether there is any way to cope with "pinning" objects.)

4.2 A General Transformation

In this section, we describe a general transformation for a wide class of dictionary algorithms. The transformation takes a dictionary implementation that works with coherent shared memory, and produces an implementation that uses multi-version memory for some of the nodes in the representation of the dictionary. We begin by describing our assumptions about the algorithm that uses coherent shared memory. Next, we describe the transformation. We conclude with a proof of correctness.

4.2.1 Assumptions

We model the data structure used to represent a dictionary as an acyclic labeled graph (where the labels are assigned to nodes). We distinguish some nodes in the graph as *leaf* nodes; a node created as a leaf cannot be changed to a non-leaf, or vice-versa.

One node is distinguished as the *anchor* node; the identity of the anchor node never changes. An edge directed out of a leaf node must be incident upon another leaf node; we do not allow edges from leaf nodes to non-leaf nodes. The label on a node represents the state of the node, including information about the keys stored at the node, the range of keys associated with each edge leaving the node, and whether each edge leaving the node points to a leaf node. We assume that all data is stored at the leaves, as in a B+-tree [Com79]; non-leaf nodes contain redundant index information to help operations find appropriate leaves.

We assume that several functions are used to implement the three dictionary operations. The functions and the assumptions we make about their behavior are specified as follows:

- The function *covers* takes a node n and a key k. If n is a leaf, then *covers* returns *true* iff n is responsible for storing information that can be used to determine whether k is in the dictionary. The subsets of the key space covered by the different leaves form a partition of the key space. We do not make any assumptions about the behavior of *covers* if n is not a leaf.

- The function *successor* takes a node n and a key k. If the label for n specifies that the range of keys associated with an edge leaving n includes k, then *successor* returns the node m, the node pointed to by the above edge. We do not make any assumptions about the behavior of *successor* if no edge described above exists.

- The function *reaches* takes a node n and a key k, and returns *true* iff n's label indicates that the leaf that covers k is reachable from n. We formally define reachable as follows. Let the function *successor*^{i}, where $i \geq 0$, be defined as follows:

$$\mathit{successor}^i(n,k) = \begin{cases} n & i = 0, \\ \mathit{successor}(\mathit{successor}^{i-1}(n,k),k) & \text{otherwise.} \end{cases}$$

  Then, k is reachable from n, if for some finite integer j, $\mathit{successor}^j(n,k)$ is the leaf that covers k. We assume that any key is reachable from the anchor.

- The function *is_in* takes a key and a leaf node that covers the key, and returns *true* iff the node's label indicates that the leaf stores the key.

- The function *is_leaf* takes a node n and returns *true* iff n is a leaf.

```
    proc find_x (k)
        n := anchor
        readlock (n)
        while ! is_leaf (n) do
            next_n := successor (n, k)
            readunlock (n)
            n := next_n
            readlock (n)
        end
        readunlock (n)
        xlock (n)
        while ! covers (n, k) do
            next_n := successor (n, k)
            xunlock (n)
            n := next_n
            xlock (n)
        end
        return (n)
    end find_x
```

Figure 4.1: *find_x* procedure.

- The function *leaf_edge* takes two nodes n and m. It assume [illegible] ıere is an edge from n to m. It returns *true* iff n's label indicates that m is a leaf.

We divide our assumptions about the coherent shared memory algorithm into two parts: assumptions about its *form*, and assumptions about its *behavior*.

Assumptions about Form

We restrict our attention to implementations of a dictionary in which the operations are implemented as follows. Each operation starts by calling a *find_x* operation, which locks and returns a leaf node that covers the specified key. There are two *find_x* operations: *find_read* and *find_write*. *Lookup* calls *find_read*, which locks the returned leaf in *read* mode; *insert* and *delete* call *find_write*, which locks the returned leaf in *write* mode. Figure 4.1 presents the implementation of the *find_x* operations.

Find_read's implementation replaces *xlock*(n) and *xunlock*(n) in *find_x*'s implementation with *readlock*(n) and *readunlock*(n), respectively; *find_write* replaces *xlock*(n) and *xunlock*(n) with *writelock*(n) and *writeunlock*(n), respectively.

After calling the *find_x* operation, the implementation of a dictionary operation executes its decisive step, which atomically either reads or updates the leaf l returned by *find_x*. In its decisive step, a *lookup* operation uses the *is_in* function to determine the result to be returned to its caller; an *insert* or *delete* operation changes l's label to reflect the addition or removal of the specified key, and may also modify the state of the graph in other ways (e.g., by adding or removing nodes and changing edges).

After executing its decisive step, a *lookup* operation returns its result. The other operations may perform more work before returning. We model this by allowing an *insert* or *delete* operation to perform a sequence of atomic updates, each involving one or more nodes. However, we view the *insert* or *delete* operation as completed as soon as it releases the *writelock* to the leaf returned by its *find_write*. Furthermore, subsequent atomic steps modify only non-leaf nodes. (The atomic updates performed after the operation's completion can be viewed as being performed by a "background process.")

We assume the dictionary uses read/write locks to ensure atomicity of the steps in the implementations of the operations. The *find_x* operation, as shown above, uses *readlocks* on non-leaf nodes, *x*locks on leaf nodes, and locks only one node at a time. After executing *find_read*, a *lookup* operation has a *readlock* on the returned leaf; it holds this lock until the result of the operation has been determined. Similarly, after executing *find_write*, an *update* operation has a write lock on the returned leaf; it holds this lock until it updates the leaf, after which it may acquire other *writelocks* on both leaves and internal nodes (e.g., for propagating splits and merges through the graph). We assume that whenever a process acquires more than one lock, it acquires and releases them in a nested fashion.

Assumptions about Behavior

The most basic assumption we make about the behavior of the coherent shared memory algorithm is that it is correct, in the sense that the dictionary operations are linearizable. We also make two additional assumptions about the *find_x* operations. First, we assume that a *find_x* operation for key k, when started at a node that reaches k, will only visit nodes that reach k. Second, we assume that a *find_x* operation for key k, if run with all other processes halted, will lock and return the leaf that covers k.

We do not assume that the coherent shared memory algorithm guarantees that all operations will terminate, since algorithms like the one presented in the previous chapter cannot make such a guarantee. However, we will assume that it is *non-blocking*, in the sense that as long as at least one operation is running, some operation will finish. Furthermore, we assume that a finite number of *insert* or *delete* operations performing their sequences of background atomic updates with all other processes blocked will eventually all complete.

```
proc find_x (k)
    n := anchor
    readpin (n)
    while ! is_leaf (n) do
        if ! reaches (n, k) then
            readunpin (n)
            readcurrent (n)
        end
        next_n := successor (n, k)
        if leaf_edge (n, next_x) then
            readunpin (n)
            n := next_n
            xlock (n)
        else
            readunpin (n)
            n := next_n
            if n has already been visited then
                readcurrent (n)
            else
                readpin (n)
            end
        end
    end
    while ! covers (n, k) do
        next_n := successor (n, k)
        xunlock (n)
        n := next_n
        xlock (n)
    end
    return (n)
end find_x
```

Figure 4.2: Transformed *find_x* procedure.

4.2.2 Transformation

Given a dictionary implementation that satisfies the assumptions described above, we modify it to use multi-version memory for the non-leaf nodes, including the anchor. The transformation requires two steps. First, we replace all occurrences of *lock* or *unlock* operations on non-leaf nodes with the corresponding *pin* or *unpin* operations. Second, we modify the coherent shared memory *find_x* implementation to incorporate multi-version memory nodes.

Figure 4.2 presents the implementation of the transformed *find_x* operations. Since the *find_x* operation can read an old version of a non-leaf node, it is possible for it to access a version of a node that does not reach the key it is trying to find. To handle this, we modify the *find_x* operation for key k as follows: if it encounters a non-leaf node that does not reach k, it does a *readcurrent* operation on the node (lines 4-6 in Figure 4.2). In Section 4.2.3, we show that the version selected by *readcurrent* always reaches k. For coherent shared memory algorithms that allow their *find_x* operation to visit "deleted" nodes, it is possible for the transformed *find_x* procedure to traverse cycles in the graph. Therefore, we issue a *readcurrent* to any node already visited (lines 15-17). In Section 4.2.3, we show that this procedure will avoid cycles.

4.2.3 Proof of Correctness

In this section, we prove that the transformed algorithm is correct. We begin with some definitions. Then we show that given a coherent shared memory dictionary algorithm that satisfies the assumptions presented above, the multi-version memory algorithm resulting from the above transformation is linearizable. We conclude by proving that the multi-version memory is *non-blocking*, in the sense that as long as at least one operation is running, some operation will finish.

Definitions

A computation of the multi-version memory algorithm can be represented by a sequence of steps, where each step is either:

- An invocation of a dictionary operation.
- A return from a dictionary operation.
- A *read* step for a node n.
- A *readcurrent* step for a node n.
- A *fail* step.
- An *update* step involving a set of non-leaf nodes $\mathcal{N}$.
- A *leaf* step involving a set of leaves $\mathcal{L}$.

Each step belongs to a particular instance of some dictionary operation. A *read* step corresponds to the entire sequence of *read* operations between a *readpin* operation and the next *readunpin* operation. A *readcurrent* step consists of the entire sequence of *read* operations between a *readcurrent* operation and the next *readunpin* operation. A *fail* step occurs if a *find_x* operation attempts to execute *successor*(n, k) where n does not reach k. (We will show that this never happens.) An *update* step corresponds to the entire sequence of *read* and *write* operations in a nested collection of *writepin* blocks. (Recall that a *writepin* block consisting of the *reads* and *writes* between a *writepin* and the subsequent *writeunpin*.) A *leaf* step corresponds to the entire sequence of *read* and *write* operations in a nested collection of *readlock* and *writelock* blocks.

A computation of the coherent shared memory algorithm can be represented by a sequence of the same kinds of steps, except that *readcurrent* steps will not appear, and the correspondence with the operations is based on *lock* and *unlock* operations instead of *pin* and *unpin*.

We say a *leaf* step belonging to a particular dictionary operation instance is a *decisive* step if it is the last *leaf* step belonging to the same dictionary operation instance. The decisive step is the one that either changes the abstract state of the structure (the set of dictionary values stored in the leaves) or determines the value to be returned by a *lookup* operation. We define an *effective step* to be an *update* or decisive *leaf* step. Also, we define an *interface step* to be an invocation of or return from a dictionary operation.

For the purpose of the proof, assume that when a *fail* step occurs in a *find_x* operation, the operation halts. This means that a dictionary operation whose *find_x* fails will not execute any *decisive* or *update* steps.

Linearizability

We show that the transformed algorithm guarantees linearizability by reduction to the coherent shared memory algorithm. Since, by assumption, the coherent shared memory algorithm guarantees linearizability of the dictionary operations, it follows from the lemma below that the transformed algorithm also guarantees linearizability.

Lemma 4 2.1 *If $\mathcal{M}$ is a computation of the multi-version memory algorithm, then there exists a computation $\mathcal{S}$ of the coherent shared memory algorithm with the same sequence of effective and interface steps.*

Proof: We prove the claim by induction on the number of effective and interface steps in $\mathcal{M}$. If the number is zero, the claim is immediate. Otherwise, let π be the last effective or interface step in $\mathcal{M}$, and let $\mathcal{M}'$ be the prefix of $\mathcal{M}$ that ends just before π. By the induction hypothesis, there exists a computation $\mathcal{S}'$ of the coherent shared memory algorithm with the same sequence of effective and interface steps as $\mathcal{M}'$. We obtain $\mathcal{S}$ from $\mathcal{S}'$ as follows. If π is an *update* step or an interface step, $\mathcal{S}$ is just $\mathcal{S}'\pi$.

Otherwise, π is a decisive *leaf* step for some key k. Let $\mathcal{F}$ be the sequence of steps executed by the *find_x* operation with argument k when started in the state after $\mathcal{S}'$, with all the other processes halted. (The *find_x* operation is *find_read* if the π belongs to a *lookup* operation, and is *find_write* otherwise.) Then $\mathcal{S}$ is $\mathcal{S}'\mathcal{F}\pi$.

It is clear that $\mathcal{S}$ as constructed above has the same sequence of effective and interface steps as $\mathcal{M}$. We must show that it is a computation of the coherent shared memory algorithm. The only difficult case is when π is a decisive *leaf* step; we must show that the *find_x* operation in $\mathcal{S}$ arrives at the same leaf as the corresponding leaf in $\mathcal{M}$. For this, we use our assumptions about the behavior of *find_x* operations in the coherent shared memory algorithm. In particular, the state of the leaves after $\mathcal{S}'$ is the same as after $\mathcal{M}'$ (by the induction hypothesis since $\mathcal{M}'$ and $\mathcal{S}'$ have the same sequence of effective steps). When the multi-version memory algorithm executes π, it has the leaf that covers k locked. Thus, after $\mathcal{M}'$ (and hence also after $\mathcal{S}'$), the leaf that covers k is same as the leaf read or written by π. By assumption then the *find_x* operation in the coherent shared memory algorithm, if run with all other processes halted starting in the state after $\mathcal{S}'$, will lock and return the leaf that reaches k. Executing π after $\mathcal{S}'\mathcal{F}$ will then give the same result as executing π after $\mathcal{M}$. ■

Liveness Properties

In this section, we prove that the transformed multi-version memory algorithm is non-blocking, in the sense that if any operations are running at any point in time, some operation will eventually complete.[4] First, we show that the multi-version memory algorithm does not allow *find_x* operations to fail (which, as mentioned earlier, would cause operations to halt). Second, we show that if the coherent shared memory dictionary is always acyclic and at least one operation is running, then some operation will complete in finite time.

Lemma 4.2.2 *A find_x operation for key k in the multi-version memory algorithm does not fail.*

Proof: Let k be the key in question for the *find_x* operation, and let $\mathcal{M}$ be a computation of the multi-version memory algorithm. We need to show that the version used by

[4] We do not allow a process to halt while in the middle of an operation. In particular, if a process has a node pinned in *write* mode and halts, then other processes that attempt to *write_pin* the node will block forever. Thus, this is a weaker notion of "non-blocking" than used, for example, by Herlihy [Her90].

the *successor* operations in $\mathcal{M}$ always reach k. We do this as follows. A *find_x* operation for key k executes a series of *read* and *readcurrent* steps R_i. If the version used in a *read* step does not reach k, a *readcurrent* step is executed. For the purposes of the proof, we consider such a *read* step and the subsequent *readcurrent* step as "sub-steps" of a single step. We prove inductively that the version used by each step reaches k.

The basic step involves the first *read* step in the *find_x* operation, which reads the anchor node. This is trivial, since by assumptions made in Section 4.2.1 and Lemma 4.2.1, the anchor reaches any key.

For the inductive step, assume that R_i reads a version A of node n_i that reaches k, and that the *find_x* operation next reads node n_{i+1} (found to be the appropriate successor of n_i in step R_i). If R_{i+1} reads a version of n_{i+1} that reaches k, we are done. If not, the *find_x* operation then executes a *readcurrent* substep R'_{i+1} for n_{i+1}. To show that the *find_x* operation does not fail, it suffices to show that the version B of n_{i+1} read by R'_{i+1} reaches k. By Lemma 4.2.1, there exists a computation $\mathcal{S}$ of the coherent shared memory algorithm with the same sequence of effective and interface steps as $\mathcal{M}$. We show that $\mathcal{S}$ can be augmented with a *find_x* operation for k that reads version A of n_i and then version B of n_{i+1}. Recall that we assumed that the coherent shared memory algorithm has the property that a *find_x* operation for k, if started at a node that reaches k, will only visit nodes that reach k. It then follows that B reaches k.

We augment $\mathcal{S}$ with a *find_x* operation for k that starts at node n_i, and reads n_i immediately after the update step U_A that writes version A. We denote this *read* step as R_A. Since the multi-version memory algorithm and the coherent shared memory algorithm use the same method for choosing successive nodes to visit in a *find_x* operation, this *find_x* will read n_{i+1} next. We simply delay this *read* step until immediately after either the *update* step U_B that writes version B, or the *read* step R_A, whichever comes later.

We must show that the *read* step for n_{i+1} inserted into S reads version B. This is trivial if the *read* step for n_{i+1} is inserted into S immediately after U_B. Otherwise, either U_B comes before U_A, or $U_B = U_A$. If $U_B = U_A$, then R_A immediately follows U_B, and the read step for n_{i+1} immediately follows R_A, so the read step for n_{i+1} reads version B of n_{i+1}. Otherwise, U_B comes before U_A. In this case, it suffices to show that no other *update* step for n_{i+1} occurs between U_B and U_A. But in $\mathcal{M}$, R'_{i+1} occurs after R_i; since a *read* step in the multi-version memory algorithm can only read a version written by a prior update step, R_i occurs after U_A. Hence, R'_{i+1} occurs after U_A. Since R'_{i+1} reads the current version of n_{i+1}, there is no update step for n_{i+1} between U_B and R'_{i+1}. Since

R'_{i+1} occurs after U_A, there is no update step for n_{i+1} between U_B and U_A.

Therefore, the *read* step for n_{i+1} inserted into S reads version B. By the assumptions made about the coherent shared memory algorithm, version B of n_{i+1} must reach k. ■

To show that the transformed multi-version memory algorithm is non-blocking, we need to introduce two concepts: ***version graphs*** and ***computation states.***

Recall that the dictionary implementation is a labeled graph. The ***version graph*** of the dictionary implementation is a directed graph that contains a separate node for each version of each node in the dictionary. An edge exists in the version graph from a version V of a dictionary node m to all versions of the dictionary node n if V contains a pointer to n. Note that by removing from the version graph all but the current version of each dictionary node, and all edges except those whose source and destination nodes are current versions, we are left with the dictionary implementation.

The *computation state* of a dictionary implementation describes not only the state of the data structures used to represent the dictionary, but also the state of the processes that are performing dictionary operations. To define the notion of a process state, we can think of each process as performing a sequence of computation steps, which we defined above. For example, a process performing a *lookup* operation must perform a finite number of *read* steps, followed by a *leaf* step. The state of a process must describe not only the sequence of computation steps the process has already performed, but also the computation steps that the process will perform in the future. We represent a process's state by a single "program counter." By convention, we set the value of a program counter to the computation step that the process will perform next. The computation state of a dictionary implementation consists of a version graph used to represent the state of the data structures and a list of program counters used to represent the state of the processes performing operations.

Lemma 4.2.3 *The multi-version memory algorithm is non-blocking. (I.e., if at least one incomplete operation is running, eventually some operation will complete.)*

Proof: Consider the computation state of the dictionary implementation. For now, assume that the sequence of nodes visited by the *read* steps in a *find_x* operation does not traverse any cycles in a version graph that does not change over time. Given this, we can show that if at least one incomplete operation is running, eventually some operation will complete.

We divide the computation state of our multi-version dictionary implementation into four cases:

1. The value of all the program counters in the computation state are set to *read* steps.

2. At least one program counter is set to a *leaf* step.

3. At least one program counter is set to an *update* step, at least one program counter is set to a *read* step, and no program counter is set to a *leaf* step.

4. At least one program counter is set to an *update* step and no program counter is set to a *read* or *leaf* step.

Case (1) is straightforward. Since the graph has finitely many nodes, does not change over time (since no program counters are set to *decisive leaf* or *update* steps), and we assume *find_x* operations do not traverse cycles, then eventually one operation will perform a *leaf* step. Data contention among *read* steps is not an issue since they all involve only *readpin* and *readcurrent* operations. Once an operation reaches a leaf, the computation state has become that of case (2).

For case (2), at least one process, which we call α, has its program counter set to a *leaf* step. This means that α has completed its *read* steps and will next perform a *leaf* step. Let $\mathcal{L}$ be the set of leaves affected by α's *leaf* step. According to our assumptions, the edges connecting leaf nodes in the coherent shared memory algorithm's implementation cannot form a cycle. By Lemma 4.2.1, the edges connecting leaf nodes in the version graph of our multi-version memory algorithm cannot form cycles either. Therefore, we can order the leaves in our version graph by using a topological sort (where the first node in the sort containing no edges leaving it). We prove by induction on the order in which the leaves in $\mathcal{L}$ appear in the topological sort that eventually some operation will perform a *decisive leaf* step and complete. For the basic step, assume $\mathcal{L}$ consists only of the first leaf in the topological sort. If we assume that read/write locks are non-blocking (and this is a reasonable assumption since the coherent shared memory algorithm is non-blocking), then eventually some operation will perform a *leaf* step affecting the first leaf in the topological sort. By Lemma 4.2.2 and the structure of the *find_x* operation, this *leaf* step must also be *decisive* since the first leaf in the topological sort does not have any edges leaving it. For the inductive step, assume that $\mathcal{L}$ contains l_i, the i'th leaf in the topological sort, and no other leaf in $\mathcal{L}$ appears before l_i in the sort. Eventually some process will perform a *leaf* step that affects the j'th leaf in the sort, where $j \leq i$. Either the step is *decisive* and a dictionary operation completes, or the process finishes the step and is now ready to perform another *leaf* step. For the second case, the set of leaves that the new step accesses includes a leaf that is at most the $(j-1)$'th leaf in the topological sort. This is because *non-decisive leaf* steps (which occur in the *find_x* operation) use the function *successor* to determine the leaf node affected by the next *leaf* operation. Since $j - 1 < i$, by induction, we conclude that eventually, some operation will perform its *decisive leaf* step and complete.

Case (3) is similar to case (1), except that some program counters have been set to *update* steps. We will prove by contradiction that after a bounded number of steps, the computation state will become that of either case (1) or (2). Assume that a computation state from case (3) will never become that of either case (1) or (2). Let $\mathcal{P}$ denote the set of processes that the *update* program counters represent (i.e., $\mathcal{P}$ represents the set of processes that will perform *update* steps). The only computation steps that can affect the state of the version graph are *decisive leaf* and *update* steps. The only way a *decisive leaf* operation can be performed is if the computation state becomes that of case (2). Also, the only way new processes can be added to $\mathcal{P}$ is if a process not in $\mathcal{P}$ performs its *decisive leaf* step which means the computation state must become that of case (2). Therefore, the only steps that can alter the state of the version graph are *update* steps, and no new processes will perform *update* steps. Furthermore, the number of processes in $\mathcal{P}$ cannot dwindle to zero, or else the computation state will become that of case (1).

This is where the contradiction appears. Since processes in $\mathcal{P}$ perform *update* steps, they only issue *writepins*. Processes not in $\mathcal{P}$ perform *read* steps; they only issue *readpins* and *readcurrents*. Thus there is no data contention between processes in $\mathcal{P}$ and not in $\mathcal{P}$. Also, processes not in $\mathcal{P}$ will not alter the state of the data structures representing the dictionary. Therefore, the processes in $\mathcal{P}$ would behave the same regardless of whether the processes not in $\mathcal{P}$ are running or blocked. In fact, if we assume that the processes not in $\mathcal{P}$ are blocked, then the behavior of the processes in $\mathcal{P}$ is identical to the behavior of the same processes running on the coherent shared memory algorithm; the only difference between *update* steps in the multi-version memory and coherent shared memory algorithms is that the multi-version memory algorithm uses *writepins* instead of *writelocks*. Since we assume in the coherent shared memory algorithm that a finite number of processes performing *update* steps with other processes blocked will eventually all complete, we can also assume that eventually the number of processes in $\mathcal{P}$ will dwindle to zero. Therefore, we have a contradiction. We conclude that the computation state must eventually become that of either case (1) or (2).

Case (4) is similar to case (3). Either some new operations on the dictionary are invoked and the computation state becomes that of case (1), or by using arguments similar to case (3), eventually all processes performing *update* steps will complete (which means that there are no more active processes performing dictionary operations).

The only thing left to prove is that *find_x* operations do not traverse cycles in a fixed version graph. (Remember, this is the key assumption we used above for case (1).) Recall that *find_x*, when accessing a node it already visited, uses *readcurrent* to read the current version of the node. Let k be the key argument specified by an ongoing *find_x* operation. If *find_x* does traverse cycles in the version graph, then the nodes in the cycle in the version graph must refer to current versions of the dictionary. Here we have a contradiction. Recall from Lemma 4.2.2 that all nodes visited by *find_x* operations must reach k, and that the definition of the function *reaches* implies that *find_x* will reach the leaf that covers k after a finite number of *successor* calls. Therefore, the cycle must not exist, and the proof is completed. ■

4.3 The Multi-Version Memory Algorithm

In this section, we build a multi-version memory algorithm by applying the transformation above to the algorithm we presented in the previous chapter. We first show that the assumptions about *form* and *behavior* described in the previous section are valid for the coherent shared memory algorithm presented in Chapter 3. We then present the resulting multi-version memory algorithm.

4.3.1 Valid Assumptions

In this section, we show that the assumptions necessary for correctness in the transformation are, for the most part, valid for the algorithm presented in the previous chapter. Discrepancies between the algorithm and the transformation assumptions can easily be fixed. The data structure used to represent the dictionary can, in fact, be viewed as a labeled graph, where each B-link tree node's state is encapsulated in its label. There is only one anchor, and it never changes. All leaf nodes stay leaves; all non-leaves stay non-leaves. All rightlinks from leaf nodes point to other leaves. (Leaves do not have downlinks.) Data is stored only at the leaves. The functions *covers?*, *successor*, *reaches?*, *find_key*, and *is_leaf?* presented in Section 3.2 perform the same tasks as the functions *covers*, *successor*, *reaches*, *is_in*, and *is_leaf* presented in Section 4.2.1.

The remaining assumptions are divided into assumptions about form and assumptions about behavior.

Assumptions about Form

All three of the dictionary operations implemented in our coherent shared memory algorithm fit the transformation's assumptions about form. We first examine the *lookup* operation, then the *update* operations.

Lookups. The procedure *ly_lookup*, shown in Figure 3.5, calls *lookup_descent*, shown in Figure 3.6, which corresponds to the *find_read* operation described above. After calling *lookup_descent*, the procedure *ly_lookup* performs its decisive operation.

Updates. The procedures *ly_insert* and *ly_delete*, shown in Figures 3.7 and 3.18, call *update_descent*, shown in Figure 3.8, which corresponds to the *find_write* operation. They then complete their decisive operations. Except for lines 9-15 in procedure *start_nodes* shown in Figure 3.13, resulting *complete_split* and *complete_merge* operations are sequences of background atomic steps which only use *writelocks*. The above mentioned lines in *start_nodes* acquire a *readlock* on the anchor. It is apparent that removing these lines will still result in a correct B-tree algorithm. However, we show below that these lines need not be removed when the transformation to a multi-version memory algorithm is applied.

Assumptions about Behavior

Our coherent shared memory algorithm also satisfies the transformation's assumptions about behavior. Specifically, the following facts are true:

- Concurrent dictionary operations in our coherent shared memory algorithm are linearizable.
- The *find_x* operation (descent phase) of all operations in our algorithm with the key k as an argument will visit only nodes that reach k.
- A *find_x* operation for key k, if run with all other processes halted, will lock and return the leaf that covers k.
- The algorithm is non-blocking.
- A finite number of *insert* and *delete* operations each performing a sequence of background atomic updates with all other processes halted will eventually complete.

The proofs of these facts have been sketched, but are not included in the thesis due to time and space constraints.

4.3.2 The New Algorithm

The transformation of our coherent shared memory algorithm presented in the previous chapter to an algorithm that uses multi-version memory for its non-leaf nodes is straightforward. We replicate the anchor and non-leaf nodes in the tree using multi-version memory; we replicate the leaves using coherent shared memory. For most of the pseudocode procedures presented in Chapter 3, the transformation simply requires changing all *lock* and *unlock* commands for the anchor and non-leaf nodes to the appropriate multi-version memory *pin* and *unpin* commands. However, the transformations for *lookup_descent* (Figures 3.6), *update_descent* (Figure 3.8), and *start_nodes* (Figure 3.13) are more complicated.

Figure 4.3 presents the pseudocode for the transformed *lookup_descent* procedure. The changes to the original coherent shared memory procedure are exactly the ones described above in the transformation of the *find_x* procedure. The check that *node* reaches *k* in lines 7-9 of Figure 4.3 is necessary, since *readpins* might select an old version that does not yet reach *k*. Because rightlinks for marked nodes "point left," *lookup_descent* must avoid cycles by issuing a *readcurrent* to the next node visited after visiting a marked non-leaf node (lines 16-17). If the next node to visit is a coherent shared memory leaf, then *lookup_descent* issues a *readlock* instead of a *readpin* (lines 18-19). The changes for the transformed *update_descent* procedure are similar to the transformed *lookup_descent*.

As explained above, lines 9-15 in the procedure *start_nodes* (Figure 3.13) are troublesome in that a *readlock* is acquired on the anchor. In our transformation, we assume the atomic update steps that occur after an *update* operation's decisive step only use *writelocks*. There are two ways to correct this problem. The first is to remove lines 9-15 from *start_nodes* and apply the normal transformation to *start_nodes* (i.e., replace the *writelock* and *writeunlock* operations with *writepin* and *writeunpin* operations). Although the seven removed lines reduce the probability that the anchor is *writelocked*, they are not necessary for correctness.

The second and more desirable way to solve the problem is to simply replace the *readlock* and *readunlock* in lines 9-15 with *readpin* and *readunpin*. Because leftmost nodes in our B-link tree are never deleted and no nodes are ever created to the left of existing leftmost nodes, the leftmost node pointers stored in old versions of the anchor are still valid (i.e., they still point to leftmost nodes of individual tree levels). Therefore, replacing the *readlock* and *readunlock* in lines 9-15 with *readpin* and *readunpin* will preserve correctness, since these lines only look up leftmost node pointers. (These lines may read an old version of the anchor. If the version contains a leftmost node pointer for

```
proc lookup_descent(k)
    % get root of tree
    readpin(ANCHOR)
    level := ANCHOR.root_level
    node := ANCHOR.leftmost_nodes[level]
    readunpin(anchor)
    % descend down tree to leaf level
    readpin(node)
    while ! is_leaf?(node) do
        % check if node reaches k
        if ! reaches?(node, k) then
            readunpin(node)
            readcurrent(node)
        else
            next := successor(node, k)
            marked? := node.marked?
            leaf? := node.level = 1 && downlink connects node and next
            readunpin(node)
            node := next
            % check if cycle might be traversed
            if marked? then
                readcurrent(node)
            else if leaf? then
                readlock(node)
            else
                readpin(node)
            end
        end
    end
    % move along leaf level to proper leaf, using readlocks
    while ! covers?(node, k) do
        next := node.rightlink
        readunlock(node)
        node := next
        readlock(node)
    end
    return node
end lookup_descent
```

Figure 4.3: Transformed *lookup_descent* procedure.

level l, then the pointer indeed points to the leftmost node in level l. If it doesn't contain such a pointer, then lines 16-18 in *start_nodes* will *writepin* the anchor and access the pointer if it exists, or create one if it doesn't.)

The transformations for all the other pseudocode procedures in the previous chapter require simply replacing all *lock* and *unlock* operations on the anchor and non-leaf nodes with the appropriate *pin* and *unpin* operations. The correctness of the new algorithm is guaranteed by the correctness of the transformation.

4.4 Summary

In this chapter, we presented *multi-version memory*, a replication scheme which loosens the semantics of *coherent shared memory* by allowing readers to access "old versions" of an object. As a result, multi-version memory implementations allow more concurrency and require less communication and synchronization than coherent shared memory schemes. Although the weaker semantics is less generally useful than coherent shared memory, it is sufficient to support a variety of B-link tree algorithms, including the algorithm presented in the previous chapter.

We presented and proved the correctness of a transformation that takes a coherent shared memory concurrent dictionary algorithm, and builds a multi-version memory concurrent dictionary algorithm. The original algorithm must satisfy a small set of assumptions. We showed that the algorithm presented in the previous chapter satisfies these assumptions, and presented a transformed multi-version memory algorithm. The correctness of the new algorithm is guaranteed by the correctness of the transformation.

Chapter 5

Performance Measurements

In this chapter, we present a series of experiments that we performed using a message-driven simulator for large scale message-passing architectures. The experiments have two purposes. The primary purpose is to examine the performance of various concurrent B-tree algorithms, including our multi-version memory algorithm presented in the previous chapter. The secondary purpose is to compare the performance and scaling properties of multi-version memory with coherent shared memory.

Any performance experiment for large-scale parallel applications must address certain key issues. *Data contention* is perhaps the most obvious issue to consider, especially for concurrent B-tree algorithms. The most critical example of data contention in the B-tree is the *root bottleneck*, which occurs during any update of the tree's anchor or root node; the root bottleneck blocks all incoming operations. The methods used to reduce the root bottleneck are the main differences between the algorithms. Unfortunately, much of the work in analyzing concurrent B-tree algorithms concentrate on data contention and ignore other important issues [BS77, Ell80, LS86, LSS87].

For example, since a concurrent B-tree heavily utilizes certain key data structures (e.g., the anchor and the root), *resource contention* could be a limiting factor in throughput. Also, as concurrent and distributed systems become larger, communication networks become more complicated. If *network latency* becomes excessive, B-tree algorithms must minimize communication to preserve performance. *Replicating* objects in memory can reduce both resource contention and communication.

We implemented our B-tree algorithms using Andrew Chien's Concurrent Aggregates (CA) language [CD90, Chi90]. CA is an object-oriented language designed to support massively parallel programs for fine-grained message-passing architectures. CA's *aggregates* are especially useful for implementing data abstractions for replicated objects. We used Chien's simulator for message-passing architectures to measure the performance of our algorithms. This simulator provides a simple approach for modeling network latency and resource contention.

The number of proposed concurrent B-tree algorithms precluded implementing every algorithm; it became necessary to pick a handful of algorithms. In Section 5.1, we present the algorithms we chose to implement along with the reasons for choosing them. Section 5.2 describes the simulator we used to measure the performance of the algorithms. Section 5.3 presents the simulation results.

5.1 B-Tree Algorithms

The algorithms we chose to implement needed to be representative of all proposed concurrent B-tree algorithms. We implemented the coherent shared memory algorithm presented in Chapter 3, since there is good reason to believe its performance is better than any other coherent shared memory algorithm proposed. We also implemented the multi-version memory algorithm presented in Chapter 4, since its performance and scaling properties are likely to be even better than its coherent shared memory counterpart. For the remainder of this chapter, we refer to the algorithms presented in Chapters 3 and 4 as "our coherent shared memory algorithm" and "our multi-version memory algorithm" respectively. Both algorithms are *link* algorithms. For purposes of comparison, we also implemented some *lock coupling* algorithms.

As discussed in Chapter 2, there are two types of lock coupling algorithms, *top-down* and *bottom-up*. Top-down algorithms perform their restructuring phases during pessimistic descents, while bottom-up algorithms perform their restructuring phases after decisive operations. We implemented two lock coupling algorithms. The Mond-Raz algorithm [MR85] is a top-down algorithm; the Bayer-Schkolnick algorithm [BS77] is bottom-up.

Since merge-at-empty strategies are more suited for database applications than merge-at-half strategies [JS89], our implemented algorithms used a merge-at-empty strategy. Since optimistic lock coupling strategies generally show better performance than pessimistic lock coupling strategies [BS77, LS86, JS90], we implemented optimistic descents for both lock coupling algorithms. Because of the improvements in performance measured by Lanin and Shasha [LSS87], both lock coupling algorithms also used *quick splits*, which writelock leaves and parents of leaves during optimistic descents for *update* operations.

To summarize, we implemented the following four concurrent B-tree algorithms:

- Our multi-version memory algorithm, which was presented in Chapter 4.
- Our coherent shared memory algorithm, which was presented in Chapter 3.
- Optimistic Mond-Raz algorithm [MR85] with coherent shared memory and *quick -splits.*
- Optimistic Bayer-Schkolnick algorithm [BS77] with coherent shared memory and *quick splits.*

5.2 The Implementation and the Simulator

We implemented our B-tree algorithms using the Concurrent Aggregates (CA) language, developed by Chien [CD90, Chi90]. CA is an object-oriented language designed for fine-grained message-passing machines. CA provides many features useful for implementing different replication schemes and modeling resource contention. A message-driven simulator for message-passing architectures, designed by Chien, measures the performance of CA programs. This section discusses how CA and the simulator model some issues that are important in concurrent B-tree performance. These issues include replication, data contention, resource contention, and network latency.

5.2.1 Replication and Data Contention

Since the simulator models message-passing architectures, we represent a replicated object as a fixed number of copies maintained in separate processors, which serve to spread load and reduce contention. CA provides a multiple-access data abstraction tool called *aggregates* that can implement replicated memory. Aggregates allow users to build a collection of homogeneous objects whose internal communication and synchronization are user-defined. By using aggregates, we can build elegant implementations for both coherent shared memory and multi-version memory. Since CA provides spinlocks, we can also implement various synchronization objects such as read/write locks and multi-version memory pins. We discuss the CA implementations of coherent shared memory and multi-version memory in more detail below.

Coherent Shared Memory

After experimenting with a variety of cache coherence protocols [AB86], we decided to implement a simple directory-based invalidation scheme. This scheme was the most efficient to implement in CA and, after preliminary measurements, judged to provide the best performance for the three coherent shared memory B-tree algorithms. To provide concurrency control, we implemented read/write locks using a distributed lock approach [Bur88, ST87]. Such locks allow more efficient implementations for readlocks than the monitor approach [Hoa74, Bri75, Dij71]. However, the synchronization necessary for writelocks grows proportionally to the number of replicated copies of the object associated with the lock.

Multi-Version Memory

We implemented multi-version memory using the approach described in Section 4.1.2. Our implementation uses version numbers to check if replicated copies are current. It also updates obsolete copies by directly copying the contents of the base copy onto replicated copies. After experimenting with a variety of implementations, we found this scheme to be most efficient for the CA-implemented multi-version memory B-tree algorithm.

Replication Factor

We represent the anchor and each node in the B-tree as a replicated object. Since some nodes are more heavily utilized than others, the number of copies maintained for each node should vary. Nodes accessed more often (nodes in the upper levels) should have more replicated copies. We used a scheme with the following number of copies for each structure:

- Leaves (generally the least frequently visited nodes) are unreplicated.
- The number of copies of the anchor is equal to the number of processes that access the B-tree.
- The number of copies of an internal node of level l is equal to the minimum of the number of copies of the anchor, and the number of copies of an internal node of level $l-1$ times a user-defined constant, which we call the *replication factor*.

It is reasonable to set the *replication factor* to the expected number of children for internal nodes, since then the total number of copies for all the nodes in each level will be about the same. Lanin and Shasha [LSS87] predict this number to be 0.69 times the maximum fanout of the tree.

5.2.2 Other Issues

Other issues that need to be addressed when implementing concurrent B-tree algorithms are resource contention and network latency. CA and Chien's simulator can model both.

Resource Contention

We simulate resource contention by setting objects in our CA code to process only one message at a time. This means that a replicated object with n copies can process only n requests concurrently, even if we discount data contention. When more than n requests are issued, the excess requests "spinwait" until a copy is freed. If the spinning continues past a user-defined amount of time, we queue the messages to alleviate saturation problems described by [And89].

Network Latency

The simulator allows the user to specify the average network latency of the modeled architecture. It assigns to each message sent by the CA program a cost equal to this latency. The simulator does not model network contention or "hot spots." The default value of this parameter is one simulated "time step." A time step is a time unit based on the modeled architecture; the simulator assigns basic operations (such as arithmetic operations, local memory accesses, etc.) a cost of one time step. By increasing the simulated network latency, we can approximate the effects that large networks can have on concurrent B-tree algorithms.

5.3 Simulation Results

We measured the performance of the four implemented algorithms using Chien's simulator. Since most B-tree applications are database-related, we investigated operation patterns where the dictionary grows slowly (a common characteristic in databases). We used both randomly selected and fixed operation patterns as well as uniformly and non-uniformly distributed keys as arguments to dictionary operations.

We divide the experiments into three categories. The first category contains the majority of the experiments; it investigates how different operation mixes affect the performance of the four algorithms. The second category compares how the coherent shared memory and multi-version memory replication schemes perform for systems with large network latencies. The final category investigates how different replication factors affect the performance of coherent shared memory and multi-version memory.

In each of the experiments, we first constructed a B-tree with 1000 dictionary elements and randomly selected keys. Unless otherwise specified, the maximum fanout of the tree was 10; the initial trees contained 4 levels. (Memory constraints in the simulator prevented building larger trees.) For most experiments, the replication factor was 0.69 times the tree's maximum fanout, rounded to the nearest integer. Unless otherwise specified, we set the network latency to the default value; every CA message was assigned a latency cost of one simulated time step. We then performed 10000 dictionary operations divided among a number of *B-tree workers*. Each B-tree worker is a process that sequentially performs dictionary operations (i.e., it waits for an operation to complete and return before starting the next operation). We measured the overall throughput (measured in dictionary operations per simulated time steps) during the 10000 operations as a function of the number of B-tree workers. Each data point shown below is the average throughput of three separate trials. The maximum number of B-tree workers was constrained by the memory requirements of the simulator; generally, the maximum number was 80-100.

5.3.1 Operation Mixes

In this section, we present the results of experiments that investigate how different operation mixes affect the four concurrent B-tree algorithms. We divide the experiments as follows:

- Experiments with random operation patterns and uniformly distributed keys as arguments.
- Experiments with random operation patterns and non-uniformly distributed keys as arguments.
- Experiments with fixed operation patterns.

Random Operations and Keys

For experiments with random operations and keys, each B-tree worker randomly selects the type of operation (*insert*, *delete*, or *lookup*) to run and the key value used as an argument. Each individual experiment fixes the probability of which operation the worker will select (e.g., 45% *lookups*, 30% *inserts*, 25% *deletes*). We measure throughput of dictionary operations as a function of the number of B-tree workers.

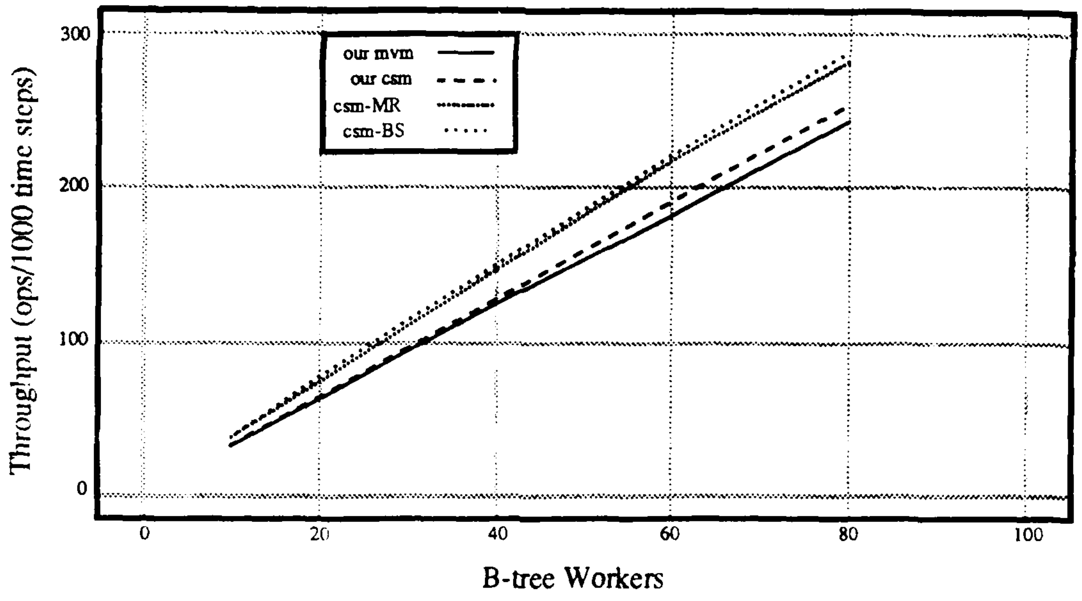


Figure 5.1: Throughput vs. B-tree workers. 100% *lookups*.

All Lookups. Figure 5.1 presents the performance of the four B-tree algorithms with only *lookup* operations. For all four algorithms, throughput behaves linearly with respect to the number of workers.

The performances of the four algorithms differ by constant factors; the two lock coupling algorithms perform better than our algorithms. The discrepancy is due to small differences in the implementation of the algorithms. The simulator assigns to each message sent by a CA program an average network latency cost. Thus CA implementations that send more messages during their descent phases will have lower throughput for a fixed number of B-tree workers. For example, the link algorithms check if the right separator of a visited node is greater than the key argument. The implementations of both algorithms in CA require extra messages for this check. By minimizing the number of messages, we can expect the performance of all four algorithms to improve by constant factors. Because the simulator is very sensitive to the number of CA messages in our implementations, we should not compare the raw performance numbers for the four algorithms. Instead, we should concentrate on the general shape of the throughput vs. number of B-tree workers curves, which indicates the general scaling properties of each algorithm.

Various Operation Mixes. Figure 5.2 presents the performance of the four algorithms with a small percentage of *update* operations. There is an 85% chance that each op-

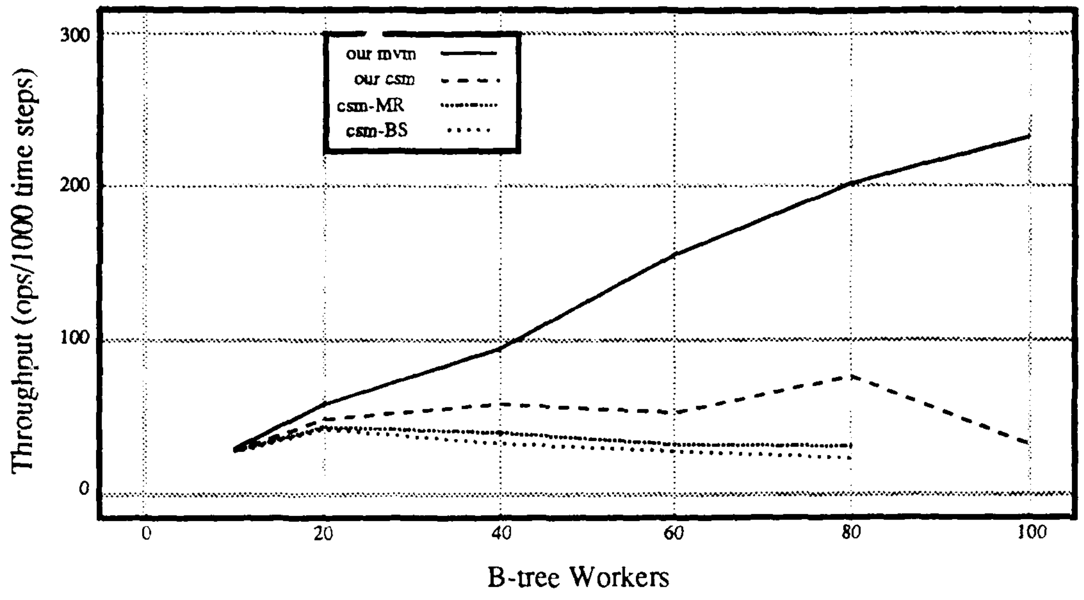


Figure 5.2: Throughput vs. B-tree workers. 85% *lookups*, 10% *inserts*, and 5% *deletes*.

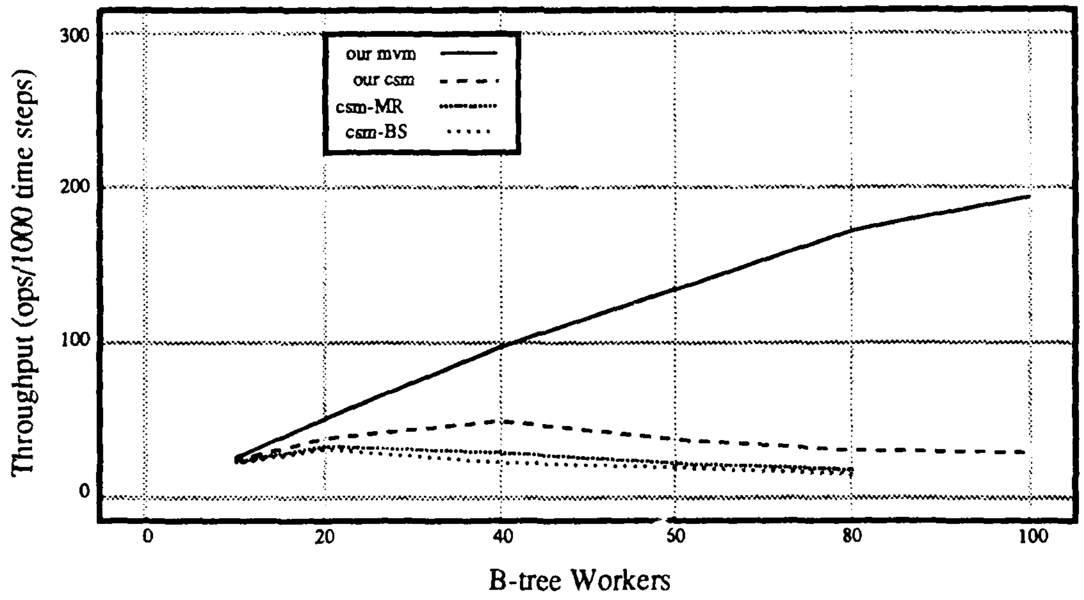


Figure 5.3: Throughput vs. B-tree workers. 45% *lookups*, 30% *inserts*, and 25% *deletes*.

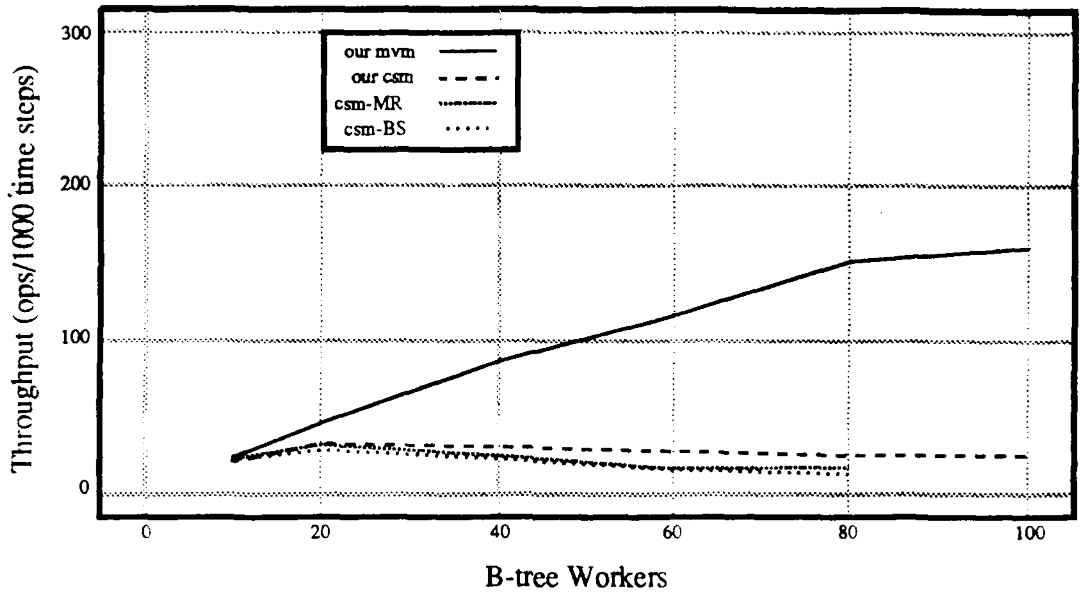


Figure 5.4: Throughput vs. B-tree workers. 5% *lookups*. 50% *inserts*, and 45% *deletes*.

eration is a *lookup*, 10% chance that it is an *insert*, and 5% chance that it is a *delete*. Figures 5.3 and 5.4 present performance measurements of experiments with even more *update* operations.

For all of the algorithms, throughput decreases as the percentage of *updates* increases. In all three experiments, the multi-version memory algorithm significantly outperforms the three coherent shared memory algorithms, especially for large numbers of workers. As discussed in Chapter 4, the multi-version memory algorithm should scale much better than coherent shared memory algorithms. Multi-version memory readers can access nodes concurrently with a writer. Data contention in the multi-version memory algorithm occurs only in the leaves (which use coherent shared memory) and when background *complete_split* and *complete_merge* operations update the same node, which is rare. Also, as the number of replicated copies of the anchor and upper-level nodes grows, the synchronization and communication needed between individual copies in multi-version memory remains relatively constant; in coherent shared memory, they grow to intolerable levels. In fact, performance starts to decrease for the coherent shared memory algorithms. The multi-version memory algorithm's performance does not exhibit such characteristics, but presumably would eventually if the number of B-tree workers were increased beyond 100. Unfortunately, constraints in the simulator restrict the experiments to model at most 80-100 workers.

Our experiments also show that our coherent shared memory algorithm performs significantly better than the lock coupling algorithms when the operation mix includes *updates*. This agrees with results from similar experiments by others [LS86, LSS87, JS90] that compare lock coupling algorithms with Lehman-Yao based coherent shared memory algorithms. Fewer writelocks, background restructuring (which lowers latency) and the elimination of lock coupling all contribute to the performance advantages of Lehman-Yao based link algorithms.

Other studies [LS86, LSS87, MR85] have found the Mond-Raz algorithm to perform much better than the Bayer-Schkolnick algorithm. Our experiments show that the performance and scaling properties of the two algorithms are very similar, and in some cases. almost indistinguishable. We explain this discrepancy in two ways.

First. unlike some of the other studies [LS86, MR85], we use the *quick split* option in both algorithms. Therefore, the chances that *update* operations require pessimistic descents are very slight. Since the pessimistic descents are the only differences between the two algorithms, we expect the differences in performance of the two algorithms to be less than the other studies.

Second, the simulator models message-passing architectures here communication between processors is relatively expensive; the simulator assigns a fixed cost to every message sent by the CA program. The fastest configurable network in the simulator still requires one simulated time step for messages to travel from sender to receiver. The top-down restructuring techniques of the Mond-Raz algorithm require more communication between B-tree nodes than the bottom-up Bayer-Schkolnick. Therefore, the Mond-Raz pessimistic descents are slower than the Bayer-Schkolnick pessimistic descents, and in some cases, cause writelocks on upper-level nodes to be held longer. Therefore, the Mond-Raz algorithm performs poorly. This phenomenon is an example of how underlying assumptions of the system architecture and network latency can significantly affect B-tree performance.

Various Maximum Fanouts. Figures 5.5 and 5.6 present performance measurements for trees with maximum fanouts of 6 and 14, respectively. The operation mix is 45% *lookups*, 30% *inserts*, and 25% *deletes*. Both experiments show lower throughput than the tree with the same instruction mix and a maximum fanout of 10 (Figure 5.3).

Having a low maximum fanout increases the number of leaves in the tree, which increases the potential concurrency in the tree. However, it also increases latency for descents and the amount of work for restructuring phases (which increases data contention). The curves in Figures 5.5, 5.3, and 5.6 illustrate the performance trade-off. If

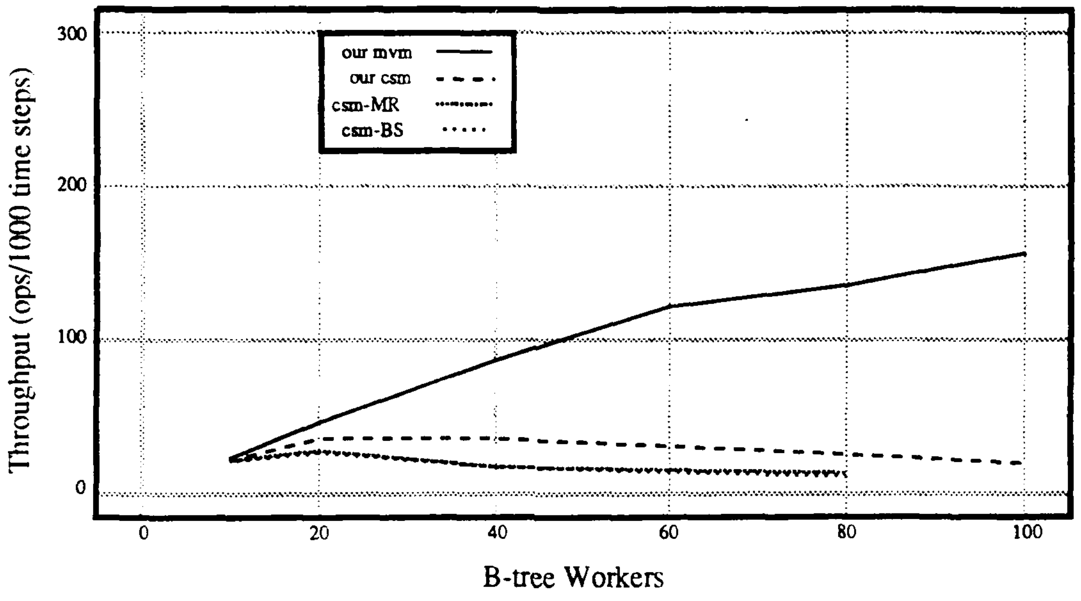


Figure 5.5: Throughput vs. B-tree workers for maximum fanout of 6.

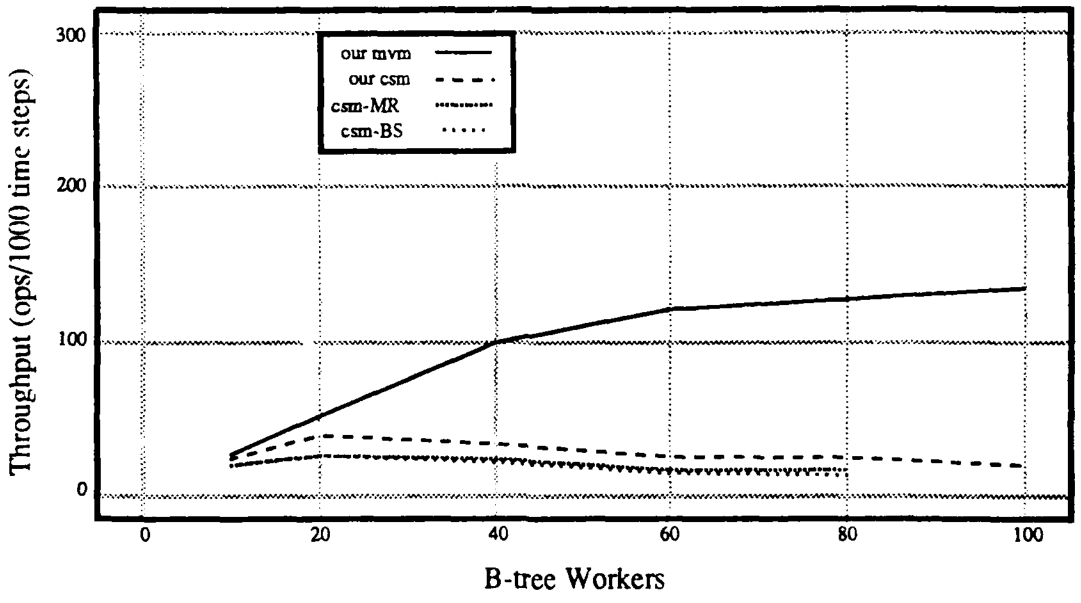


Figure 5.6: Throughput vs. B-tree workers for maximum fanout of 14.

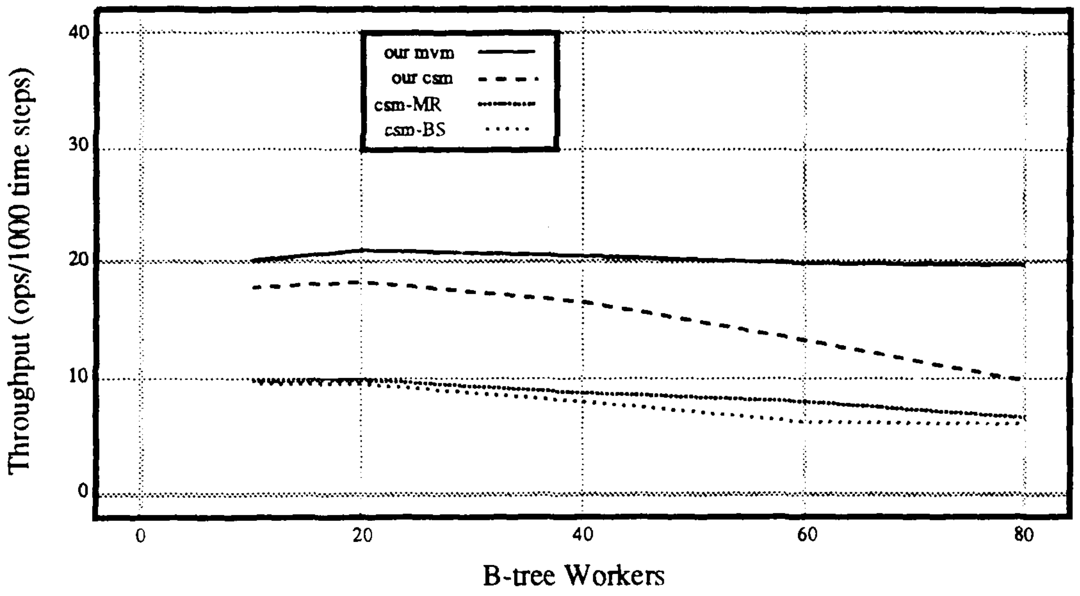


Figure 5.7: Throughput vs. B-tree workers. Incrementing localized keys.

the choice of maximum fanout produces either a "short, fat tree" or a "tall, thin tree," performance will suffer for each of the algorithms.

Localized Keys

For these experiments, we allow the B-tree workers to choose operations randomly as before, but select the key arguments non-uniformly. Each worker maintains a variable whose value is a key. Approximately half the time, the workers choose the keys randomly. For the other half, they set the key argument to the variable value and increment (or decrement) the variable. By localizing the initial values of the variables, we can increase contention among concurrent B-tree operations.

Figure 5.7 and 5.8 shows the performance measurements taken when the variable is incremented and decremented, respectively. Both increased data and resource contention caused by the highly localized key selection contribute to significantly lower throughput than in previous experiments. Performance for all the algorithms starts to degrade much earlier than in previous experiments. In fact, Figure 5.7 shows almost no measurable speedup for any of the four algorithms. In both experiments, our coherent shared memory algorithm performs significantly better than the lock coupling algorithms, and the multi-version memory algorithm performs significantly better than any of the coherent shared memory algorithms.

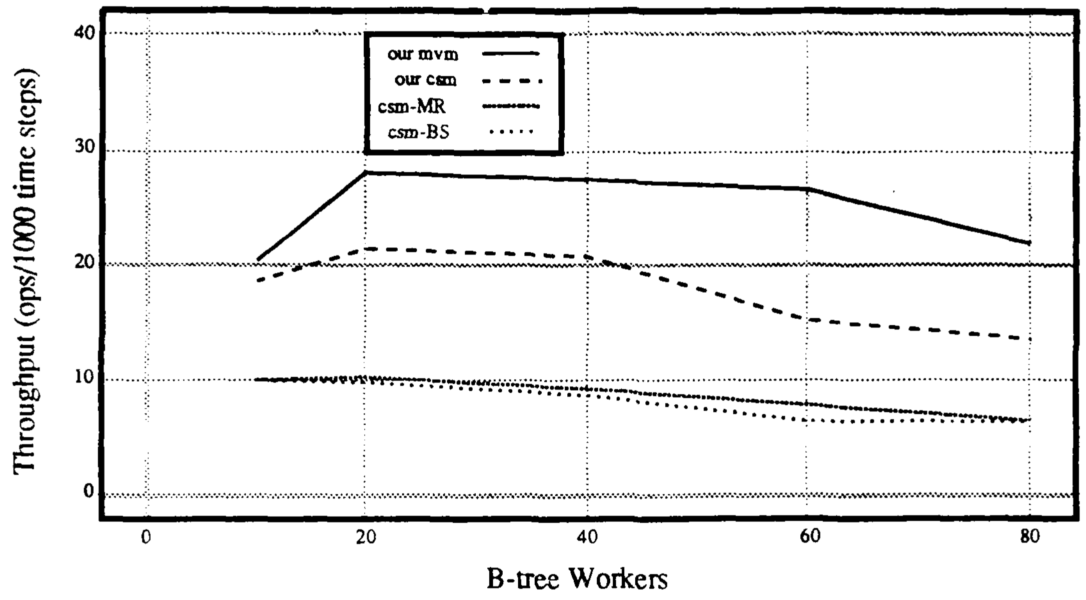


Figure 5.8: Throughput vs. B-tree workers. Decrementing localized keys.

Our two algorithms perform significantly better in Figure 5.8 than in Figure 5.7. This is because when a node is split, the *right* half is shifted to the newly created node, and the *left* half stays in the same node. Process overtaking sometimes forces our algorithms' descents to visit nodes to the *left* of the proper path. Descents for our two algorithms traverse fewer rightlinks as a result of process overtaking when workers decrement their variables than when they increment their variables. Therefore, decrementing the variables results in higher throughput than incrementing the variables.

Priority Queue

The previous experiments allow the B-tree workers to choose their operations randomly. However, in some B-tree applications, the operation pattern of processes accessing the tree is consistent and predictable. One such application is the *concurrent priority queue.*

The *priority queue* is a dynamic set of dictionary elements that supports the operations *insert* and *extract_min* (among others). The *extract_min* operation returns and deletes the item in the set with the smallest key value. A large variety of parallel algorithms use priority queues, e.g., multiprocessor scheduling and parallel best-first search of state-space graphs [Win84, Nil80, Pea84, KRR88]. Concurrent priority queues support concurrent operations; some implementations allow the *extract_min* operation to extract not just the element with the minimum key, but also an element with a "small" key.[1]

Implementing a priority queue in a Lehman-Yao type B-link tree is straightforward. Since the anchor stores pointers to leftmost nodes, an *extract_min* operation can trivially find the leftmost leaf (and thus return and delete the element with the smallest key) without a descent. For most applications using a concurrent priority queue, the processes accessing the queue exhibit a fairly consistent pattern. After an *extract_min*, a process performs several other operations (such as *inserts*) whose key arguments are localized around the extracted key. Then it performs another *extract_min*, and so on.

We implemented the *extract_min* operation on both of our algorithms. We did not use the two lock coupling algorithms because, unlike in our algorithms, implementing *extract_min* would have required a descent phase to reach the leftmost leaf. (Maintaining in the anchor a pointer to the leftmost leaf is difficult to implement, especially when the leftmost leaf is deleted.) Therefore comparing the performance of our link algorithms with the lock coupling algorithms would not be fair.

We designed our experiment as follows. After building an initial tree of 1000 keys with a maximum fanout of 10, the B-tree workers perform 10000 total operations. Each B-tree worker performs an *extract_min* followed by five *inserts*. The key arguments to the *inserts* are randomly chosen from a range of values localized around the extracted key. Afterwards, another *extract_min* is performed, and the pattern repeats. We measure throughput as a function of the number of B-tree workers.

Figure 5.9 presents the results of the experiment. Because all *extract_min* operations and many localized *insert* operations must access the unreplicated leftmost leaf, saturation problems caused by resource contention are sharper than in any previous experiment, especially for data points with large numbers of B-tree workers. The performance of the multi-version memory algorithm is still significantly better than the coherent shared memory algorithm, since multi-version memory allows more concurrency for *insert* descents and restructuring operations.

Summary

In this section, we presented performance measurements for the four B-tree algorithms for a variety of operation patterns and key selection schemes. We discovered that our coherent shared memory algorithm performed better than the two lock coupling algorithms, and that our multi-version memory algorithm had significantly better performance and

[1]Huang [Hua90] discusses *concurrent priority queues* in much more detail.

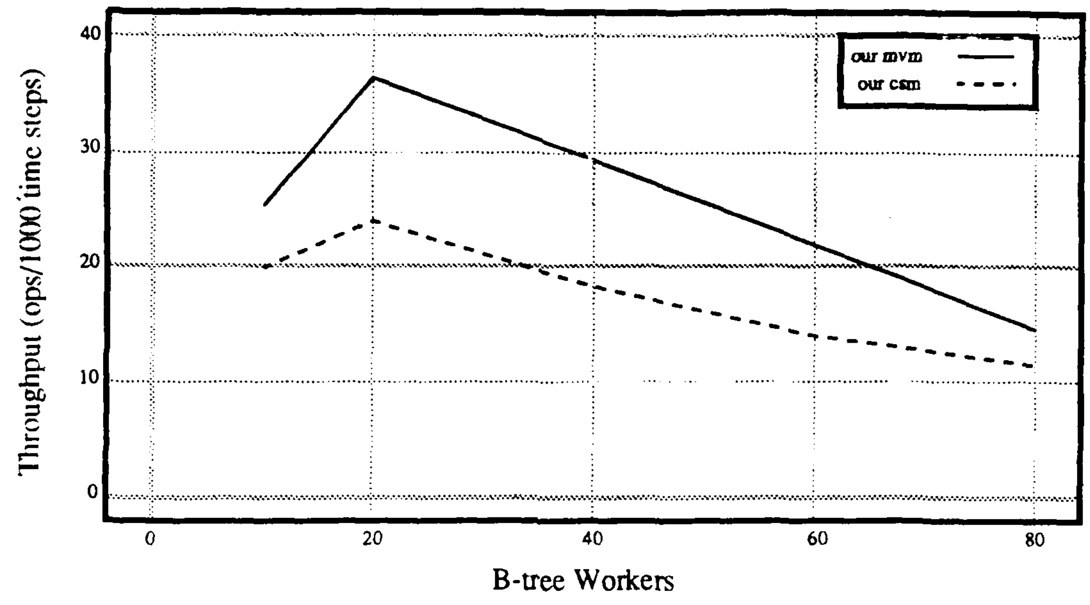


Figure 5.9: Throughput vs. B-tree Workers. Priority queue implementation.

scaling properties than the other algorithms. Replication, resource contention, and network latency had a significant effect on B-tree performance, especially for the two lock coupling algorithms. The weaker semantics of multi-version memory allows more concurrency and less communication and synchronization, thus allowing higher throughput.

5.3.2 Large Network Latency

In this section, we describe the results of an experiment designed to compare the performance of multi-version memory and coherent shared memory algorithms for systems with high network latency. We compared our multi-version memory and our coherent shared memory algorithms as a tool for the comparison. After building an initial tree of 1000 keys and with a maximum fanout of 10, B-tree workers perform 10000 total operations. We used randomly selected operations and uniformly distributed keys. The operation mix was 45% *lookups*, 30% *inserts*, and 25% *deletes*. We set the simulator's network latency to 16 simulated time steps, sixteen times that of the previous experiments. Throughput was measured as a function of the number of B-tree workers. Except for network latency, the parameters of this experiment are identical to the parameters for the experiment whose results are presented in Figure 5.3.

Figure 5.10 presents the results of the experiment. Since network latency is much greater than previous experiments, the throughput values are much lower than that of

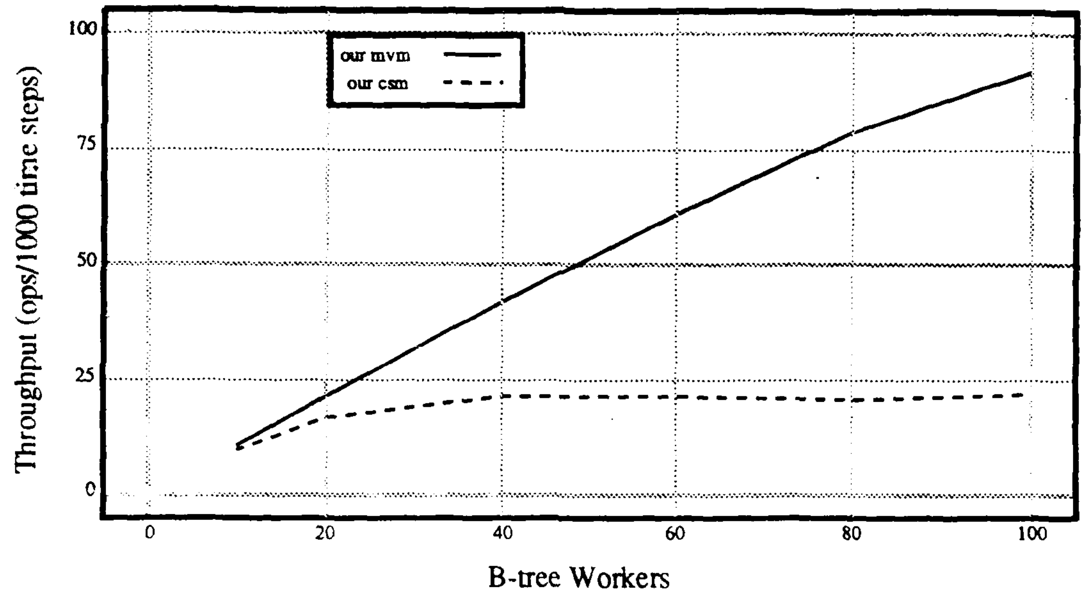


Figure 5.10: Throughput vs. B-tree workers. Slow network.

Figure 5.3. The throughput curve for the multi-version memr algorithm is almost linear, and does not experience the saturation characteristics oı previous experiments. This suggests that for this experiment, the large network latency is the overwhelming factor in limiting performance. However, the coherent shared memory algorithm exhibits significant performance degradation at around 40 workers. We can attribute this to the expensive communication and synchronization required to implement coherent shared memory, especially when an update has been performed. Multi-version memory, on the other hand, does not incur these costs. The results of this experiment suggest that for systems with large network latencies and for applications that can use its looser semantics, multi-version memory is much more suitable than coherent shared memory.

5.3.3 Replication Factor

In this section, we describe the results of an experiment designed to compare the performance of multi-version memory and coherent shared memory schemes as a function of the number of replicated copies that have to be managed. As suggested in Chapter 4, the synchronization and communication necessary for maintaining coherent shared memory grow with the number of replicated copies. Multi-version memory requires less synchronization and communication than coherent shared memory.

We compared our multi-version memory and coherent shared memory algorithms.

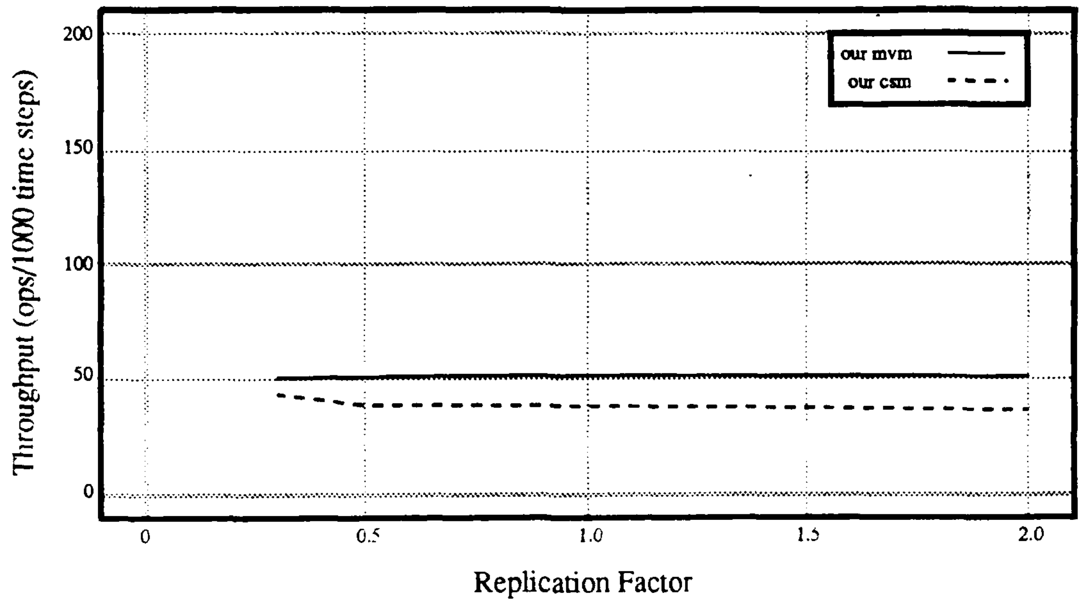


Figure 5.11: Throughput vs. replication factor. 20 B-tree workers.

After building an initial tree of 1000 keys and with a maximum fanout of 10, B-tree workers perform 10000 total operations. We used randomly selected operations and uniformly distributed keys. The operation mix was 45% *lookups*, 30% *inserts*, and 25% *deletes*. We set the simulator's network latency to the default value (i.e., one simulated time step). After fixing the number of B-tree workers, we measured throughput as a function of the replication factor.

For coherent shared memory, we expect low throughput for both very low and very high replication factors. Low replication factors limit the amount of concurrency in the B-tree by limiting the total number of replicated copies of nodes at each level. High replication factors require expensive synchronization and communication to keep large numbers of replicated copies coherent. For multi-version memories, we also expect low throughput for very low replication factors. The effect of high replication factors on multi-version memory is less clear. As explained in Chapter 4, multi-version memory implementations can do away with the costs in keeping copies coherent. However, increasing the number of replicated copies also slows down the rate at which newer versions reach the replicated copies. Thus as the replication factor increases, so does the chance that readers access old versions of B-tree nodes. This may cause more rightlinks to be traversed, which increases latency and decreases throughput.

Figures 5.11 and 5.12 presents performance results for 20 and 100 workers, respec-

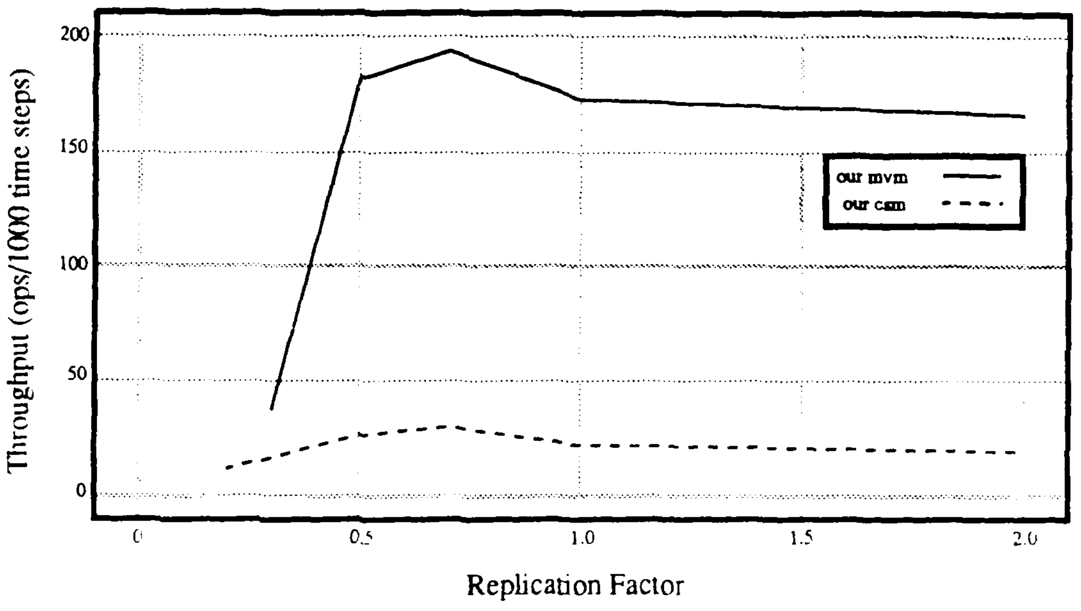


Figure 5.12: Throughput vs. replication factor. 100 B-tree workers.

tively. For 20 workers, performance for multi-version memory is virtually constant as the replication factor is varied, and performance for coherent shared memory decreases slowly as the replication factor increases. This is the result of the excess synchronization and communication needed in coherent shared memory. We do not see performance degradation for low replication factors. This is because of the small number of B-tree workers in the experiment: the low replication factors do not affect the overall concurrency of the experiment.

For 100 workers, low replication factors adversely affect performance for both algorithms. Also, the performance of both memory schemes decreases as the replication factor increases. It is likely that the multi-version memory algorithm's performance declines with higher replication factors because of an increase in the number of old versions accessed by readers; however, we do not have detailed enough data to verify this hypothesis. For the range of replication factors measured in this experiment, the multi-version memory algorithm significantly outperforms its coherent shared memory counterpart.

From this experiment we conclude that maintaining large numbers of replicated copies in both multi-version memory and coherent shared memory adversely affects performance, but for different reasons. For coherent shared memory, too many replicated copies causes synchronization and communication to grow beyond acceptable levels. For some multi-version memory implementations, too many replicated copies may cause the copies to contain old versions, which affects performance. However, the magnitude of this effect obviously depends on the application.

5.4 Summary

In this chapter, we discussed the performance of various concurrent B-tree algorithms. The algorithms include our multi-version memory and coherent shared memory link algorithms as well as two lock coupling algorithms. We used a message-driven simulator to model the algorithms' performances on a large scale message-passing architecture. Our simulations accounted for the effects of data and resource contention, replication, and network latency. The results show that our multi-version memory algorithm presented in Chapter 4 has the best performance and scaling properties. Multi-version memory allows for higher replication and concurrency while decreasing synchronization and communication.

Chapter 6

Conclusions

In this thesis, we investigated concurrent B-tree algorithms. We presented two new algorithms, one of which uses a novel replication scheme called *multi-version memory* to improve performance significantly. We showed in the previous chapter that our two algorithms perform much better than other proposed concurrent B-tree algorithms and that multi-version memory significantly improved the scaling properties of B-trees. In this chapter, we summarize the contributions of the thesis and discuss directions for future work.

6.1 Contributions

The contributions of the thesis are threefold:

- It presents the *multi-version memory* replication abstraction.
- It proposes two new concurrent B-tree algorithms, one using *coherent shared memory*, and the other modified to use multi-version memory.
- It compares the performance of various concurrent B-tree algorithms, including the algorithms proposed above.

The multi-version memory abstraction offers memory replication with higher concurrency and scaling properties than coherent shared memory. The cost of this improvement is a looser semantics that is less generally useful. However, as described in Chapter 4, multi-version memory is useful for a variety of dictionary algorithms. We can view multi-version memory as a specific example of a more general idea, *software cache management*. In such a scheme, the user can specify with an application the semantics of hardware caches. While others have proposed managing caches in software [BMW85, SS88, CSB86, BCZ90], they do not change the semantics of the replicated memory.

Allowing the user to specify in software the semantics of hardware caches fits naturally into the object-oriented programming style based on inventing application-specific abstract data types, such as that advocated by Liskov and Guttag [LG86]. Complex cache management algorithms can be encapsulated in the implementations of the abstract data types, and can be changed depending on the access patterns of the application.

The two concurrent B-tree algorithms we propose are both based on the Lehman-Yao algorithm as modified by Sagiv, and use ideas suggested by Lanin and Shasha. They perform better than any other proposed B-tree algorithm. The multi-version memory algorithm, in particular, exhibits much better performance and scaling properties than coherent shared memory algorithms.

The performance measurements of Chapter 5 suggest that replication, resource contention and network latency play an important role in determining performance for concurrent B-tree algorithms. In some cases, issues commonly ignored by existing work on concurrent B-trees dramatically affect measured performance. F example, Lanin and Shasha [LS86] and Lanin, et al. [LSS87] found the optimistic Mond-Raz top-down lock coupling algorithm [MR85] to perform significantly better than the *optimistic* Bayer-Schkolnick bottom-up lock coupling algorithm [BS77]. Taking network latency in the underlying architecture into consideration, the performance differences between Mond-Raz and Bayer-Schkolnick algorithms are sometimes not very significant.

6.2 Future Work

We categorize directions for future work into two general areas: the multi-version memory abstraction and concurrent B-tree analysis.

6.2.1 Multi-Version Memory

The *multi-version memory* abstraction is clearly a useful replication tool for concurrent B-trees and any dictionary data structure that satisfies the set of constraints presented in Chapter 4. Future work should include investigating other applications that can use multi-version memory to improve performance. For example, some iterative relaxation algorithms [Bau78] do not require processes to obtain the most recent version of certain values. Any version will guarantee correctness and termination. Another group of applications that may benefit from multi-version memory is parallel algorithms that use speculative concurrency [Hal88]. While up-to-date information may help such algorithms allocate resources efficiently, it is not essential for correctness. If versions kept by replicated copies remain relatively recent, multi-version memory may improve performance due to its ability to reduce synchronization and communication between independent processes.

We can also build a multi-version memory *spin monitor*, an idea by William Weihl which provides the same synchronization tools as conventional monitors [Hoa74, Bri75, Dij71]. Instead of descheduling processes that wait on a condition variable, spin monitors allow processes to loop around the condition using *readpins*. Spin monitors might be especially useful for applications where the number of waiting processes is large; they avoid the rescheduling overhead in conventional monitors.

The experiments in the previous chapter used an implementation of multi-version memory that propagates new versions directly from the base copy to the replicated copies. Chapter 4 outlined a variety of implementation alternatives, such as invalidation schemes and dynamic adjustment of the number of replicated copies. Future work should include a study comparing the performance of different multi-version memory implementations.

The more general idea of software cache management (of which multi-version memory is a specific example) is another important area to focus future work. As parallel and distributed systems become larger, it may become necessary for the user to specify complex cache management algorithms. Implementing software cache management in existing parallel architectures will be difficult, especially for abstractions such as multi-version memory, which require processes to *pin* cache entries. Existing or proposed architectures would have to be modified to support general software cache management. Investigating the usefulness of software cache management for large parallel applications will help decide whether modifying hardware design is either worthwhile or feasible.

6.2.2 Concurrent B-Trees

Although the simulator used for the performance measurements in Chapter 5 takes into account many issues such as replication, resource contention, and network latency, it does not make precise measurements. For example, it treats network latency in a very naive fashion. (An average cost is assigned to every CA message without any regard to locality or network contention.) Resource constraints prevented experiments for very large scale simulations (such as thousands of B-tree workers). Implementing and measuring the concurrent B-tree algorithms in a more accurate and more efficient simulator should provide better insight into the behavior of these algorithms.

For example, a parallel simulator designed by Dellarocas and Brewer [DB90] models the behavior of parallel programs for a diverse variety of architectures. It measures network latency not by a fixed average cost, but by comprehensive models for different network topologies. Because the simulator is not a "cycle-by-cycle" simulator, but one that allows individual threads to run for some variable number of cycles, resource constraints are much less stringent than the simulator used in this thesis. Furthermore, the user can specify efficiently "without cost" the types of performance characteristics to be measured. Such a simulator would be very useful in generating more detailed information about the algorithms. (Unfortunately, Dellarocas and Brewer's simulator was completed too late to be used in this thesis.)

The results of such simulations should help us better understand the performance of concurrent B-trees when issues such as resource contention, replication and network latency are factored in. This would allow us to derive an accurate, comprehensive analytical model for predicting performance. Johnson and Shasha [JS90] have developed an analytical model for concurrent B-tree algorithms that takes into account data and resource contention. A model that includes replication and network latency would be even more helpful.

A second area of future work for concurrent B-tree algorithms concerns parent pointers for B-link tree nodes, an idea proposed in Section 3.7. These pointers can reduce the overhead required for performing *update* operations.

Bibliography

[AB86] J. Archibald and J.-L. Baer. Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. *ACM Transactions on Computer Systems*, 4(4):273–298, November 1986.

[ABM89] Y. Afek, G. Brown, and M. Merritt. A Lazy Cache Algorithm. In *Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures*, pages 209–222, July 1989.

[And89] Thomas E. Anderson. The Performance Implications of Spin-Waiting Alternatives for Shared-Memory Multiprocessors. Technical Report 89-04-03, Department of Computer Science, University of Washington, April 1989.

[AVL62] G. M. Adel'son-Vel'skiĭ and E. M. Landis. An Algorithm for the Organization of Information. *Soviet Mathematics Doklady*, 3:1259–1263, 1962.

[Bau78] Gérard M. Baudet. Asynchronous Iterative Methods for Multiprocessors. *Journal of the Association for Computing Machinery*, 25(2):226–244, April 1978.

[Bay72] R. Bayer. Symmetric Binary B-trees: Data Structure and Maintenance Algorithms. *Acta Informatica*, 1:290–306, 1972.

[BCZ90] J. Bennett, J. Carter, and W. Zwaenepoel. Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence. Technical Report Rice COMP TR89-98, Rice University, 1990.

[BM72] R. Bayer and E. M. McCreight. Organization and Maintenance of Large Ordered Indexes. *Acta Informatica*, 1(3):173–189, 1972.

[BMW85] W. C. Brantley, K. P. McAuliffe, and J. Weiss. RP3 Processor-Memory Element. In *Proceedings of the International Conference on Parallel Processing*, pages 782–789, 1985.

[Bri75] Per Brinch Hansen. The Programming Language Concurrent Pascal. *IEEE Transactions on Software Engineering*, SE-1(2), June 1975.

[BS77] R. Bayer and M. Schkolnick. Concurrency of Operations on B-trees. *Acta Informatica*, 9:1–22, 1977.

[Bur88] Michael Burrows. Efficient Data Sharing. Technical Report 153, University of Cambridge Computer Laboratory, December 1988.

[CD90] A. Chien and W. Dally. Concurrent Aggregates (CA). In *Proceedings of the Second Symposium on Principles and Practice of Parallel Programming*, pages 187–196. ACM, March 1990.

[Che86] D. Cheriton. Problem-oriented Shared Memory: A Decentralized Approach to Distributed System Design. In *Proceedings of the 6th International Conference on Distributed Computing Systems*, pages 180–197, May 1986.

[Chi90] A. Chien. *Concurrent Aggregates: An Object-Oriented Language for Fine-Grained Message-Passing Machines.* PhD thesis, MIT, 1990.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* The MIT Press and McGraw-Hill, 1990.

[Com79] D. Comer. The Ubiquitous B-Tree. *ACM Computing Surveys*, 11(2):121–128, June 1979.

[CSB86] D. Cheriton, G. Slavenburg, and P. Boyle. Software-Controlled Caches in the VMP Multiprocessor. In *Proceedings of the 13th International Symposium on Computer Architecture*, pages 366–374, June 1986.

[DB90] Chris N. Dellarocas and Eric A. Brewer. The Parallel Architecture Simulator. MIT PSG Design Note draft, 1990.

[DC88] W. J. Dally and Andrew Chien. Object Oriented Concurrent Programming in CST. In *Proceedings of the Third Conference on Hypercube Computers*, pages 434–9. Pasedena, California, 1988. SIAM.

[DCF+89] W. J. Dally, Andrew Chien, Stuart Fiske, Waldemar Horwat, John Keen, Michael Larivee, Rich Lethin, Peter Nuth, Scott Wills, Paul Carrick, and Greg Fyler. The J-Machine: A Fine-Grain Concurrent Computer. In *Information Processing 89, Proceedings of the IFIP Congress*, pages 1147–1153. IEEE, August 1989.

[Dij71] E. W. Dijkstra. Hierarchical Ordering of Sequential Processes. *Acta Informatica*, 1(2):115–138, 1971.

[DS85] W. J. Dally and C. L. Seitz. The balanced cube: A concurrent data structure. Technical Report 5174:TR:85, Caltech, June 1985.

[Ell80] C. S. Ellis. Concurrent Search and Inserts in 2-3 Trees. *Acta Informatica*, 14(1):63–86, 1980.

[Hal88] R. Halstead, Jr. Parallel Computing Using Multilisp. In J. Kowalik, editor, *Parallel Computation and Computers for Artificial Intelligence*, pages 21–49. Kleuwer Academic Pub., 1988.

[Her89] M. Herlihy. Concurrent B-trees without Locking. Draft, October 1989.

[Her90] M. Herlihy. A Methodology for Implementing Highly Concurrent Data Structures. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 197–206, March 1990.

[Hoa74] C. A. R. Hoare. Monitors: An Operating System Structuring Concept. *CACM*, 17(10):549–557, October 1974.

[Hua90] Qin Huang. An Evaluation of Concurrent Priority Queue Algorithms. Master's thesis, MIT, August 1990.

[HW90] M. Herlihy and J. Wing. Linearizability: A COrrectness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[JFS85] M. Jipping, R. Ford, and R. Schultz. On the Performance of Concurrent Tree Algorithms. Technical Report 85-07, University of Iowa, 1985.

[JFSW90] M. Jipping, R. Ford, R. Schultz, and B. Wenhardt. On the performance of concurrent tree algorithms. Submitted for publication, 1990.

[JS89] T. Johnson and D. Shasha. Utilization of of B-trees with inserts, deletes, and modifies. In *ACM SIGACT/SIGMOD/SIGART Symposium on Principles of Database Systems*, pages 235–246. ACM, 1989.

[JS90] T. Johnson and D. Shasha. A Framework for the Performance Analysis of Concurrent B-tree Algorithms. In *Proceedings of the 9th ACM Symposium on Principles of Database Systems*, April 1990.

[KL80] H. T. Kung and P. L. Lehman. Concurrent Manipulation of Binary Search Trees. *ACM Transactions on Computer Systems*, 5(3):354–382, 1980.

[Knu73] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, 1973.

[Kor83] H. F. Korth. Locking Primitives in a Database System. *Journal of the ACM*, 30(1):55–79, January 1983.

[KRR88] Vipin Kumar, K. Ramesh, and V. Nageshwara Rao. Parallel Best-First Search of State-Space Graphs: A Summary of Results. In *National Conference of Artificial Intelligence*, pages 122–127, August 1988.

[KW82] Y. S. Kwong and D. Wood. A New Method for Concurrency in B-trees. *IEEE Transactions on Software Engineering*, SE-8(3):211–222, May 1982.

[Lam79] L. Lamport. How to Make a Multiprocessor that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28:690–691, 1979.

[LG86] B. Liskov and J. Guttag. *Abstraction and Specification in Program Development.* MIT Press, 1986.

[LS86] V. Lanin and D. Shasha. A Symmetric Concurrent B-Tree Algorithm. In *1986 Proceedings Fall Joint Computer Conference*, pages 380–386, November 1986.

[LSS87] V. Lanin, D. Shasha, and J. Schmidt. An Analytical Model for the Performance of Concurrent B-tree Algorithms. NYU Ultracomputer Note 311, NYU Ultracomputer Lab, 1987.

[LY81] P. L. Lehman and S. B. Yao. Efficient Locking for Co urrent Operations on B-Trees. *ACM Transactions on Database Systems*, 6 :650–670, December 1981.

[MR85] Y. Mond and Y. Raz. Concurrency Control in B+ Trees Using Preparatory Operations. In *Proceedings of the 11th International Conference on Very Large Data Bases*, pages 331–334, August 1985.

[Nil80] Nils J. Nilsson. *Principles of Artificial Intelligence.* Tioga Press, 1980.

[Pea84] Judea Pearl. *Heuristics - Intelligent Search Strategies for Computer Problem Solving.* Addison-Wesley, 1984.

[PN85] G. F. Pfister and V. A. Norton. "Hot Spot" Contention and Combining in Multistage Interconnection Networks. *IEEE Transactions on Computers*, C-34(10):943–948, October 1985.

[Sag86] Y. Sagiv. Concurrent Operations on B-Trees with Overtaking. *Journal of Computer and System Sciences*, 33(2):275–296, October 1986.

[Sal85] B. Salzberg. Restructuring the Lehman-Yao Tree. Technical Report BS-850-21, College of Computer Science, Northeastern University, January 1985.

[SG88] D. Shasha and N. Goodman. Concurrent Search Structure Algorithms. *ACM Transactions on Database Systems*, 13(1):53–90, March 1988.

[SS88] D. Shasha and M. Snir. Efficient and Correct Execution of Parallel Programs that Share Memory. *ACM Transactions on Programming Languages and Systems*, 10(2):282–312, April 1988.

[ST83] Daniel D. Sleator and Robert E. Tarjan. A Data Structure for Dynamic Trees. *Journal of Computer and System Sciences*, 26(3):362–391, 1983.

[ST87] William E. Snaman, Jr. and David W. Thiel. The VAX/VMS Distributed Lock Manager. *Digital Technical Journal*, (5):29–44, September 1987.

[Wed74] H. Wedekind. On the selection of access paths in a database system. In J. W. Klimbie and K. L. Koffeman, editors, *Database Management*, pages 385–397. North Holland Publishing Company, 1974.

[Win84] Patrick H. Winston. *Artificial Intelligence*. Addison-Wesley, 2nd edition, 1984.

[WW90] William E. Weihl and Paul Wang. Multi-Version Memory: Software Cache Management for Concurrent B-Trees (extended abstract). In *Proceedings of the 2nd IEEE Symposium on Parallel and Distributed Processing*, pages 650–655, December 1990.

OFFICIAL DISTRIBUTION LIST

DIRECTOR	2 copies
Information Processing Techniques Office	
Defense Advanced Research Projects Agency (DARPA)	
1400 Wilson Boulevard	
Arlington, VA 22209	
OFFICE OF NAVAL RESEARCH	2 copies
800 North Quincy Street	
Arlington, VA 22217	
Attn: Dr. Gary Koop, Code 433	
DIRECTOR, CODE 2627	6 copies
Naval Research Laboratory	
Washington, DC 20375	
DEFENSE TECHNICAL INFORMATION CENTER	12 copies
Cameron Station	
Alexandria, VA 22314	
NATIONAL SCIENCE FOUNDATION	2 copies
Office of Computing Activities	
1800 G. Street, N.W.	
Washington, DC 20550	
Attn: Program Director	
HEAD, CODE 38	1 copy
Research Department	
Naval Weapons Center	
China Lake, CA 93555	